AD-A049 484   ARMY ELECTRONICS COMMAND FORT MONMOUTH N J   F/G 9/2

COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE FINAL REPORT. --ETC(U)

SEP 77   E LIEBLEIN, J WAGNER, H STONE

UNCLASSIFIED   ECOM-4530   NL

1 OF 2

AD
A049484

OF 2

9484

11

Research and Development Technical Report

ECOM-4530

COMPUTER FAMILY ARCHITECTURE SELECTION COMMITTEE - FINAL REPORT, VOLUME V - PROCEDURE FOR AND RESULTS OF THE EVALUATIONS OF SOFTWARE BASES OF THE CANDIDATE ARCHITECTURES FOR THE MILITARY COMPUTER FAMILY

E. Lieblein
J. Wagner
Center for Tactical Computer Sciences
Communications/Automatic Data Processing Laboratory

H. Stone
University of Massachusetts

A. Clearwaters
Naval Underwater Systems Center

J. RODRIGUES
SOFTECH INC.

September 1977

DISTRIBUTION STATEMENT

Approved for public release:
distribution unlimited.

ECOM

US ARMY ELECTRONICS COMMAND FORT MONMOUTH, NEW JERSEY 07703

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 14 ECOM-4530	2. GOVT ACCESSION NO. 9 Research and development technical rept.,	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) 6 Computer Family Architecture Selection Committee Final Report, Volume V. Procedure For and Results of the Evaluations of Software Bases of the Candidate Architectures for the Military Computer Family.		5. TYPE OF REPORT & PERIOD COVERED 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) 10 E. Lieblein, A. Clearwaters ~~NUSC~~ J. Wagner, H. Stone, (~~[redacted]MASS~~) J. Rodrigues ~~(Softech Inc.)~~		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center For Tactical Computer Sciences (CENTACS) DRSEL-NL-BC Fort Monmouth, N.J. 07703		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 16 1L7 62701 AH92~~B109~~ 17 B1
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 11 Sep[redacted]77 13. NUMBER OF PAGES 110 Pages 12 115p.
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) CENTACS DRSEL-NL-BC Fort Monmouth, N.J. 07703		15. SECURITY CLASS. (*of this report*) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

This report consists of a summary and nine (9) volumes. It is the result of joint Army/Navy work. The complete report may be separately published by the Naval Research Laboratories.

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

Computer Family Architecture, Support Software, Host-Target Concept, Software Tools.

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

One of the primary reasons for adopting the architecture of an existing successful computer family as the architecture for a future Military Computer Family is the potential utility of the existing support software base. As such, the evaluation of the amount of support software for each of the three finalist architectures was deemed to constitute a very important factor in the overall selection process. This paper details the software evaluation process utilized as well as the quantitative results of that process.

037 620 JOB

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TABLE OF CONTENTS

1. INTRODUCTION AND SUMMARY

At the third meeting of the Computer Family Architecture (CFA) Selection Committee held on 17-20 February 1976, it was agreed that final selection of the CFA would be based not only on the architecture features that had been evaluated by that date but also on test programs, the existing software bases, licensing costs, and architecture subsetability/expandability. Subcommittees on each of these areas were formed. In addition, a selection methodology subcommittee was formed and asked to develop a decision procedure to integrate the results of the various subcommittees.

On 20 February 1976, the Software Evaluation Methodology Subcommittee was requested to develop the criteria for evaluating the software bases of the CFA finalists and the procedure for obtaining a quantitative evaluation. This document describes this methodology. Section 2 of this volume describes the problem in greater detail. Section 3 contains the technical approach which includes a general approach to the structuring of a software base. Section 4 discusses the evaluation criteria utilized to rank the three finalist architectures in terms of software base, and Section 5 contains the results of the evaluation. The remainder of this section is a summary of the methodology utilized to evaluate the software bases and a summary of the results.

Table I depicts one of the main results of the software study. It shows a comparison of the software bases and deficiencies for the three finalist architectures. In order to put these figures in context, it is felt that a brief description of the methodology utilized to obtain these figures (described in later sections) is necessary.

The first part of the methodology was to compile and describe all tools that could in any way be utilized for the development of DoD software. Next, each voting committee member was asked to vote on the applicability of each tool as it applied to his/her activity. The results of this balloting were compiled and the list of tools was generated in order of points received. Those tools which received less than a threshold of 1,000 points were excluded from further consideration because it was felt that it would not pay for DoD to develop tools with such a low relative score in applicability.

The next step in the process was to have the manufacturers determine which tools they marketed. This information was audited by an independent agent as to its accuracy. Next, software tools which meet the tool criteria of the report referenced above, and are marketed by other than the architecture manufacturers were compiled. Both of these lists of tools were combined into lists, for each architecture, of the tools available and not available.

In order to convert these lists into dollar figures, an estimate of the development cost of each tool was needed. Each manufacturer was requested to estimate the size of each tool he had available in source lines of code. Utilizing productivity figures and standard costs for a man-year of effort, a development cost for each tool was derived. The costs were compiled for each architecture, to determine the software base and deficiencies for each architecture as shown in Table I.

TABLE 1 - SOFTWARE BASE AND DEFICIENCY COMPARISON

	BASE	DEFICIENCY
IBM	32,269K	9,595K
DEC	22,220K	19,130K
INTERDATA	15,360K	25,970K

Next, it was deemed necessary to estimate the number of years it would require the government to make each architecture totally nondeficient in support software. In order to do this, the three lists of deficient tools, one each for each architecture, were taken and ordered in terms of the applicability scored. Next, these lists were reordered slightly in terms of a PERT sequence. In other words, if tool X scored lower in applicability than tool Y, but the development of X before Y would ease the development of Y or other tools, then tool X would be placed higher on the list.

These lists were then converted into schedules based upon anticipated investments of $1 million, $2 million and $3 million a year and realistic development periods. From these nine schedules, Table 2 was created depicting the years it would take to bring each architecture to nondeficiency.

This data was then factored into two distinct cost models described elsewhere to obtain best estimates of the life cycle costs for each of the three architectures.

TABLE 2 - YEARS TO CORRECT DEFICIENCIES

	DEVELOPMENT DOLLARS		
	1M	2M	3M
IBM	10.5	5.5	4.5
DEC	20	11	8.5
INTERDATA	26	15	10

2. STATEMENT OF THE PROBLEM

One of the main reasons for adopting the architecture (i.e., instruction set) of an existing successful computer family as the architecture for a future Military Computer Family (MCF) is the potential utility of the existing software base. This was supported almost unanimously at the third CFA Selection Committee meeting. Use of the term "existing" does not imply that MCF interest is limited only to the software that exists on 23 August 1976 since it is expected that each existing software base will continue to expand. In fact, a premise of the MCF program is that if the CFA selected is the same as that used in an existing and very successful computer family, that CFA will continue to exist and be successful and its software base will continue to expand independently of the MCF program. Therefore, the existence of a CFA software base that has a higher measure of immediate utility will be taken as an indication of the potential utility of the future software base of that CFA.

There is a significant amount of software associated with each of the architecture finalists. However, a mere listing of such software would not be adequate for relative ranking of the finalists. The problem then is to (1) define what is meant by the terms "software base", (2) determine what should and can be measured, and (3) develop a methodology for assessment of the value of a given software base that facilitates a relative quantitative ranking of the four CFA finalists by 23 August 1976.

3. TECHNICAL APPROACH

a. Domain of the Software Base

In the commercial world there is a good deal of applications software that is transportable across applications. Examples are payroll systems, inventory systems, and general ledger (account/billing) systems which are developed and supported as market items. The military world of tactical and strategic systems has not been able to achieve such transportability, primarily due to the disparity among applications, the highly complex time-critical and sensor-oriented functions involved, and the relative immaturity of its systems (i.e., it is not easy to isolate commonality at the early stages of evolution of a complex requirement, however, after the development of three or four similar systems, common functions should begin to emerge).

After a brief attempt to find commonality among existing military applications software failed, it was the consensus of the subcommittee that applications software should not be considered within the domain of the software base. Only support software will be included. Support software includes that software used for producing, modifying, analyzing, and testing a computer-based system. There will be no distinction made between the support software that runs in the operational target environment (e.g., executive systems and data-base management systems) and the support software that only runs in the development (support/host) environment (e.g., compilers and simulation systems).

The section describing the software base will serve as a more precise definition of the term support software. Specifically, any software defined there will be considered to be support software.

b. The Military Software Development Environment

The evaluation of support software cannot be completed without some knowledge of the way in which such software will be used in the development of military computer-based systems. While it is not the intent of this report to delineate a complete approach to the software development environment for the MCF, the report cannot be completed without a preliminary consideration of this important area. For example, if it were decreed by DOD that all MCF software had to be developed on a CDC-6600 host environment and then transferred to the MCF target hardware, then it would probably be the case that the support software base would be a non-issue in the selection of the CFA; i.e., the importance of the CDC-6600 support software base would far outweigh the importance of the software bases of the CFA finalists, and the CDC-6600 software base would apply equally as well to each CFA finalist.

(1) Host-Target Concept

(a) General

In the past, far too many small to medium scale military computer systems were developed using the computer that is to go into the final operational system as its own software development tool. Many of the consequences of this approach were disastrous since these development environments are virtually devoid of the very broad spectrum of powerful support software tools that now exist on larger computers. (Such tools will be described in detail in the remaining sections of this report). This section describes an approach to software development in which programs are developed on one computer, the host, for operation on another computer, the target. The basic rationale for this approach is to optimize the host computer configuration and its support software for the development task and to allow the target computer configuration to be optimized for the objectives of the system being developed. Such an approach is especially applicable to the development of software systems for military tactical applications whose computer configurations are inadequate for effective use of high level language compilers and other relatively large and powerful support software tools.

The host-target approach has the potential of minimizing the cost of support software because many support software tools are independent of the target computer and also has the potential of reducing development costs of an environment optimized for the development task.

However, the host-target approach introduces a significant problem in the testing activities of development if the host and target computers are of different architecture families. It is desirable to conduct program tests on the host computer in order to minimize the complexity of the development process and to benefit from the test support software of the host system, but differences between the host and target architectures may affect the validity of such testing.

The following sections describe target-independent development activities, target-dependent development activities, development tools affected by the host/target approach, and the implications for the MCF.

(b) Target-Independent Development Activities

Many of the activities of a software development project are, at least, partially independent of the target computer architecture. This includes activities such as configuration management where the activity itself is relatively independent as well as activities such as editing a source program where, though the activity product may be target-dependent, the support software tools used can be independent of the target computer. Some examples of target-independent activities are:

(1) Planning
(2) Requirement Determination
(3) System Design
(4) Design Validation
(5) Program Library Management
(6) Documentation
(7) Configuration Management

Many of these activities are now being performed with the aid of computer-based support tools. The possibility of improving the development process by use of such tools has been recognized and there is a trend toward even greater use of computers to support these activities. The culmination of this trend seems likely to be the totally integrated software development system, a relatively large, integrated data base system containing a complete description of the development project including (1) requirement specifications, (2) component representations (design, program code, test data, documentation, etc.) (3) information about the project activities, and (4) relations among the requirements, component representations, and activities. Support software tools will be associated with the data base to perform generation, analysis, transformation, and reporting of the project descriptive data. A key design goal for the support system is to automate maintenance of the completeness, currency, and integrity of the project descriptive data. This will have a significant positive effect on the visibility of project progress, product reliability, and maintenance costs. Totally integrated software deveopment systems are now being developed. Some are at least partly operational and are being refined and extended. It is clear that such support tools are expensive to develop and require large-scale computer systems (with extensive memory and I/O capabilities) for effective operation. These requirements make it highly unlikely that such software support systems will ever be developed for operation on the smaller military tactical computers or their commercial counterparts. (Even the more traditional large-scale software development systems cannot be supported on such small computers.)

The potential benefits of using such an integrated support software system and the high cost of developing it constitute strong arguments for centralizing software development support on one or a few types of host computer even though the software is being developed for operation on one of a variety of target computers.

(c) Target-Dependent Development Activities

Several activities of a software development project are dependent on the target computer in which the developed programs will operate. A host-target approach must provide for this dependency. The most significant of these dependent activities is testing. Program translation also has some dependent aspects.

Final software testing and some categories of system testing require operation in an environment as close to the application as possible. A host-target approach does not provide a rationale for relaxing this requirement; therefore, it is necessary to provide for some target computer use during development. One of the objectives of the testing of individual program components and groups of related programs is to provide a basis for confidence that the system and final software tests can be completed without extensive changes. Therefore it is beneficial for the environment of program testing to closely resemble the operational environment also, although this requirement is not as demanding as the system test requirement. If the host computer architecture differs from that of the target computer, then the amount of final software testing that is possible on the host computer will be limited to that which can be done through simulation (and possibly emulation) of the target computer. Simulation/emulation may be a reasonably effective substitute for the target system in a good portion of the final software testing if the target is a small-scale computer system. However, if the target system is, e.g., a command and control system that employs a large scale computer system with extensive I/O operations and if the host and target architectures are incompatible, then software/system testing will be severely hampered because the host will waste too much time in its mimicking of the target architecture to be an effective test vehicle.

Additional target dependent activities are compilation, assembly, and link-editing. However, these dependencies are significantly different from the testing dependency. They result primarily from the fact that most existing program translators have been built to operate on and produce code for the same computer architecture. This dependency does not result from the intrinsic requirements of the translation process. Basically, program translation is a data processing operation which is relatively independent of the computer on which it is performed and which can be most efficiently performed in the relatively large memory spaces of host computers.

(d) Host-Target Development Tools

This section discusses the impact of the host-target approach on three classes of support tools:

(1) Compilers and Assemblers
(2) Target Simulators or Emulators
(3) Host-Target Communication

One possible approach to achieving the transfer from test operation on the host computer to operation on the target computer is to program in a high level language and to use two compilers, one producing host computer code and the other producing target computer code. However this approach is not generally applicable within the current state-of-the-art because most currently available programming languages are not sufficiently standardized and machine-independent for trouble-free transfer of programs from one machine architecture to another and because the requirements for many military computer-base systems cannot be met without machine-dependent programming. Where requirements permit, this approach should be considered but it should not be expected to be uniformly applicable.

Another approach to the problem of transfer from host to target computer is to use cross translators and a simulator or emulator of the target computer operating on the host computer. The cross translator operates on the host computer and translates source programs to target machine object programs. Target computer programs may be tested by interpretation on the host computer through use of a target computer simulator program or emulator microprogram. It is possible to faithfully imitate the logical characteristics of the target computer by simulation or emulation but usually it is not feasible to match the target computer's timing characteristics. While emulation of the target computer by microprogramming the host computer may provide a closer approximation of the target computer's timing characteristics as well as significantly greater throughput than simulation, the number of available computer models suitable for emulation of a variety of target computers is quite limited at the present time. Since the host system will most likely be commercial hardware (due to cost and availability factors), the general purpose emulation capability cannot be counted on.

Testing conducted on the target computer may also be supported by tools operating on the host computer. Some of the possible types of such support are:

(1) Transmission of Programs from Host to Target
(2) Storage of Test Data Recordings
(3) Test Data Analysis and Reduction
(4) Simulation of the Application Environment
(5) Monitoring and Controlling Test Operation

One of the most critical basic test support capabilities is automated communication between the host and target computers for transmission of programs, data, and test control commands. If the computer hardware is suitable, the most effective test interface would permit the host computer to directly address the target computer memory and to interrupt its CPU.

(e) Summary of Host-Target Concept

Applying a host-target approach to the software development process involves both advantages and disadvantages. The advantages proceed from the possibility of using a standardized, comprehensive, and coordinated set of support software tools. The disadvantages proceed from the added development task of transferring the development programs from the host to the target computer. The advantage of using a host computer optimized for software development while allowing the target computer to be optimized for its particular application is significant. However, if the host and target computers are of different architecture families, then the disadvantages become severe, especially with regard to the use of the host in the development of the larger military computer systems such as command and control systems. On the other hand, if the host and target computers use the same architecture, and if the commercial host has a powerful software development system, then a major portion of the software development and testing may be accomplished on the host and the programs may be transferred to the target without difficulty. Therefore it is concluded that the tactical software development process may be greatly enhanced through the implementation of a host-target concept where both host and target have the same architecture. This has significant implications with respect to the final choice of architecture for the MCF.

(2) Software Development Activities

In this section a model will be presented that aided in the quantitative assessment and relative ranking of the CFA finalists with respect to their software bases. Committee members were asked to indicate the relative importance of the various items (software tools) in the software base model by the assignment of weights to software tool types. In order to assign weights in a meaningful way, it was important to understand not only what the tool does but also in which part of the software development cycle it is used, i.e., which activity it supports. In this way, a committee member was able to place more weight on a tool that aids an activity in the software development cycle that he considered to be more important. To aid this process, each tool description contained a reference to the applicable activity of the software development cycle. In this section, the software development cycle activities will be delineated.

The software development process is partitioned into the following major activities:

(1) Analyze Requirements
(2) Design Software
(3) Build System Tests
(4) Build and Unit-Test Software
(5) Integrate and System Test
(6) Maintain System

Figure 1 illustrates these software development activities and their interrelationships. The lines with arrows and dots at each end indicate both an output (result) and feedback path. The "tag" on each such line separates each action by a slash, i.e., output/feedback. Now, each activity will be described.

USED AT:

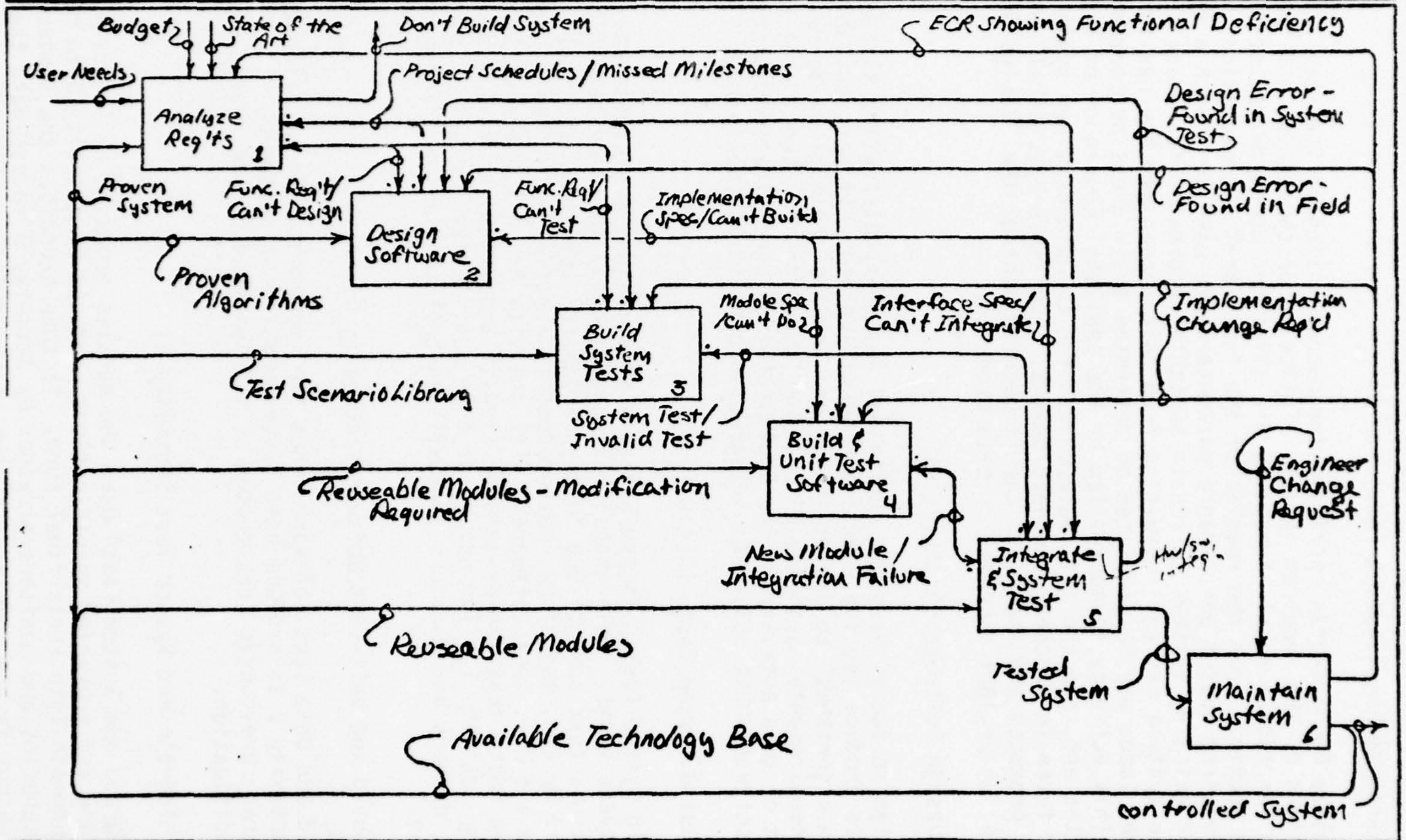
AUTHOR: C. Young
PROJECT: Software Evaluation
NOTES: 1 2 3 4 5 6 7 8 9 10

DATE: 3/15/76
REV:

☒ WORKING
□ DRAFT
□ RECOMMENDED
□ PUBLICATION

READER DATE

CONTEXT: Top


Budget
State of the Art
Don't Build System
ECR Showing Functional Deficiency
User Needs
Analyze Req'ts 1
Project Schedules / Missed Milestones
Design Error - Found in System Test
Proven System
Func. Req't/ Can't Design
Design Software 2
Func. Rqt/ Can't Test
Implementation Spec/Can't Build
Design Error - Found in Field
Proven Algorithms
Build System Tests 3
Module Spec /Can't Do
Interface Spec/ Can't Integrate
Implementation Change Req'd
Test Scenario Library
System Test/ Invalid Test
Build & Unit Test Software 4
Engineer Change Request
Reuseable Modules - Modification Required
New Module / Integration Failure
Integrate & System Test 5
Hw/sw integ'n
Reuseable Modules
Tested System
Maintain System 6
Available Technology Base
controlled System

NODE: A0

TITLE: Develop Software Systems

NUMBER:

FIGURE 1

In 3.3, the tools that support a single activity as well as those tools that are common across activities will be described briefly.

(a) Analyze Requirements (Activity 1)

"Analyze Requirements" performs the decomposition of the user needs into the functions of the required system. Following decomposition and the development of a functional model, functions are allocated to hardware, software, firmware, and people. Then the results of the functional decomposition, allocation and the required system performance parameters are used to search a descriptive catalog of existing systems to locate suitable candidates for reuse or modification. The systems (if any) resulting from this search and any new functions that must be developed may be simulated to determine their gross performance characteristics. This activity is controlled by the analysts' knowledge of the current state of the art, and the available budget for the proposed new system. If a decision is taken to proceed with development, a software functional design specification is produced and recorded, and is used to control the "Design Software" activities. A project schedule is also produced.

(b) Design Software (Activity 2)

"Design Software" uses the functional design specification allocated to software to produce the implementation specification. It has available a library of "proven algorithms" to assist in design, and has the ability to respond to the "Analyze Requirements" activity with a "can't design" or "can't meet schedule." The output of this activity is the implementation specifications that will control the software unit build and integration activities (Activities 4 and 5).

(c) Build System Tests (Activity 3)

"Build System Tests" is the activity that designs and constructs the system acceptance tests. Note that it is controlled by the same set of functional specifications that control the "Design Software" activity, and that it is unconstrained by and, therefore, may proceed in parallel with "Design Software" and "Build and Unit Test Software." The activity has a library of previously constructed tests that are presumably tied to subsystems that will be reused as directed by Activity 1. The output of this activity is the set of system test scenarios, drivers and monitors that will control Activity 5, "Integrate and System Test."

(d) Build and Unit-Test Software (Activity 4)

"Build and Unit Test Software" uses the implementation specifications produced by Activity 2 to produce unit tested modules of software. It has available a library previously constructed of modules that can be reused with or without modification.

(e) Integrate and System Test (Activity 5)

"Integrate and System Test" uses the modules produced by Activity 4 and the interface and subsystem specifications produced by Activity 3 to bind the system components into their final form. It then exercises the system using the test scenarios and monitors provided by Activity 3 to validate the system. Its final output is the completed system released to the maintenance, distribution and configuration control activity (Activity 6). Integration and/or test failures are reflected back to the appropriate design, test production or software production activities.

(f) Maintain System Activity (Activity 6)

The final development activity, "Maintain System" is primarily a clearinghouse and control center for the reception, evaluation and control of engineering change requests. These are routed to the appropriate activity for re-implementation, re-design, or re-analysis. The maintenance function distributes configuration-controlled systems to the users, and releases the results of the development effort into the available technology data-base.

c. Structuring of the Software Base

(1) General

The software base will be structured, in part, by partitioning the software tool types according to the specific development activities they support. However, this approach may conceal that part of the software base that supports the operation of such tools and does not clearly indicate those tools that support all activities. To make such software visible, the software base will be structured further through a layered approach that will provide insight into the relationships among software base components. Further structuring of the software base according to sub-activities supported, disciplines supported--e.g., management, engineering (design and program development), testing, simulation, and documentation--or according to evaluation categories--e.g., functionally, reliability, performance, cost, and maintainability--was considered but deemed impractical.

There are at least five distinct layers associated with an operational computer system (Figure 2) and three layers of software that support the software development process.

Layer 0 represents the bare computer hardware including items such as processors, channels, main storage, mass storage, bulk I/O, archival storage, hardware monitors, terminals, sensors, and communications interface devices. Layers 1 through 3 "reside" on the hardware and collectively provide the virtual machine capability that is necessary to support layer 4 the development of applications software. In the following paragraphs, layers 1 through 3 are described along with short descriptions of the tools that reside in these layers and the relationship of such tools to the various development activities discussed in 3.2.2.

For more detailed descriptions, the reader is referred to Appendix A.

(2) Layer 3: Functional Support Tools

Layer 3 contains those tools that provide direct support to the six software development activities of Figure 1 and are the tools with which the applications software developer has the greatest interaction. Layer 3 tools will be related to the specific development activities they support.

(a) Layer 3 Tool Types That Support Activity 1 (Analyze Requirements)

The types of tools that are directly applicable to requirement analysis are listed below:

SOFTWARE BASE STRUCTURE

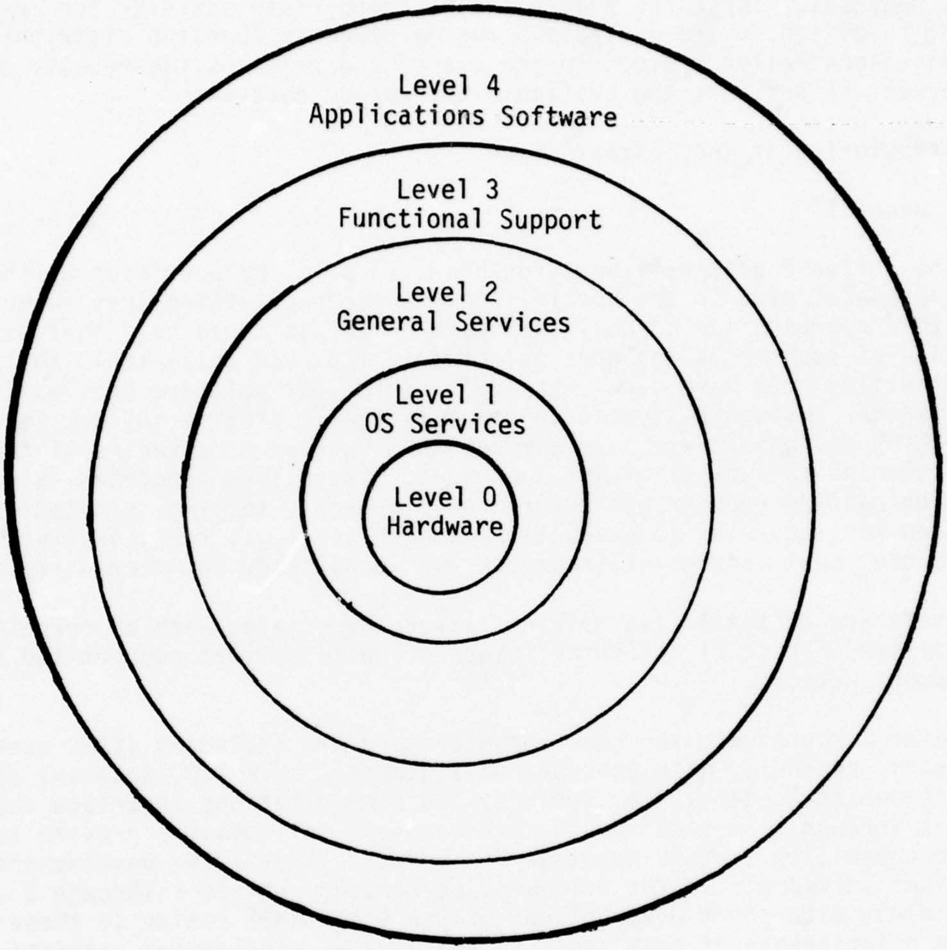


FIGURE 2

(1) General Purpose System Simulators

Allows a user to construct a computer model of a real or proposed system and to perform simulation experiments to determine the behavior of the model under various operational conditions.

(2) System Description Languages & Analyzers

Assist system analysts in describing the functional characteristics of a system and in validating the consistency and completeness of a functional decomposition.

(b) Layer 3 Tool Types That Support Activity 2 (Design Software)

The types of tools that are directly applicable to the software design are listed below:

(1) Computer System Simulators

Similar in nature to the general purpose simulator except that its basic building blocks represent real computer system components whose modeled behavior approximates the throughputs, capacities and access times achievable on the modeled equipments.

(2) Data Base Design Aids

Assist data base designers in grouping data elements into logical record classes and in determining the relationships among logical record classes implicit in either the nature of the data or the usage of the data.

(3) Data Dictionary Systems

Assist data base designers in managing the data definition activities.

(c) Layer 3 Tool Types That Support Activity 3 (Build System Tests)

The types of tools required to support system test construction are listed below:

(1) Test Data Generators

Create data files for testing and validating computer programs.

(2) Test Data Auditors

Compare data files against specifications and produce reports of discrepancies and/or compliance.

(3) Test Case Design Advisors

Analyze programs written in a high level language and present the results of that analysis in a form suitable to assist test case designers in the selection of test data.

(4) Test Instrumenters and Analyzers

Instrument modules under test so as to collect data characterizing the behavior of the module.

(d) Layer 3 Tool Types That Support Activity 4 (Build and Unit - Test Software)

The types of tools that are required to support the program development and unit-test activity are listed below:

(1) Assemblers

Allow programs to be coded in a symbolic language in which statements generally correspond to a single machine instruction.

(2) Compilers

Translate programs written in a high level language into either relocatable object code acceptable to a linker or assembly language acceptable to an assembler.

(3) Linkers

Combine the text produced by separate invocations of compilers and assemblers ("object modules") into executable code strings ("load modules" or "core images") that can be loaded into the computer's main storage and executed without further pre-processing.

(4) Debugging Aids

Assist the programmer in locating the sources of program errors that have been discovered during unit testing, usually by giving him some control over the execution of the module under test that is external to the normal program code.

(5) Module Libraries & Change Control Systems

Provide computer controlled maintenance of groups of related source modules (programs), object modules (the output of assemblers and compilers), and load modules (the output of linkers).

(6) Performance Monitors

Assist the programmer in quantifying the resource consumption characteristics of a program and in iso!ating performance-critical areas.

(7) Standards Enforcers

Allow source programs to be examined automatically and checked for conformance to installation-defined standards of format, content, and usage.

(8) Preprocessors/Reformatters

Assist programmers in producing well-structured and readable programs by allowing the introduction of structured programming constructs into source programs for languages that do not have them, and by automatically controlling indentation, the placement of comments, etc., to produce readable listings.

(e) Layer 3 Tool Types That Support Activity 5 (Integrate and System Test) and Activity 6 (Maintain System)

There are nc unique layer 3 tools that exist to support these activities. The tools that were listed for activities 1 through 4 are generally applicable to activities 5 and 6 at layer 3. Most of the tools used in practice that are specifically oriented to activity 5 are special-purpose, e.g., test environment tools (emulators, hot benches, system integration lab support, virtual machines), test drivers (drive test scenario against system; use test monitor to collect, record, and evaluate test results) and special performance monitors.

(3) Layer 2: General Support Services

The primary function of layer 2 tools is to provide a framework of common services that will allow the outputs of third layer functions to be stored, retrieved and inter-communicated. Second layer functions should be usable for common purposes across different third layer functions, and should serve to hide (where possible) differences between first layer and third layer functions. Layer 2 tool types provide general support to all of the software development activities. The layer 2 tool types are summarized as follows:

(1) Data Base Management System

Allow the user of a computer system to define the contents of and the logical relationships between collections of data items that represent some useful abstraction of a real-world phenomenon (tactical command and control system, the modules and documentation of a system of computer programs) without being concerned with the physical mechanics of storing, locating, and retrieving items or groups of items.

(2) PERT/CPM Systems

Assist managers in planning and controlling project activities.

(3) Project Estimation Systems

Assist in the development of work breakdown structures and related performance standards for use in estimating project resource requirements.

(4) Documentation Aids

Assist in the preparation and maintenance of documentation about the modules of a system. Aids most relevant to a program development environment include text processing systems, flowchart construction languages and automatic flowcharters.

(5) Data Manipulation Utilities

Allow the system user to alter the format and content of data files independent of the logical significance of the data fields involved. While this category includes the standard media conversion utilities such as card-to-tape, tape-to-printer, etc., these have not been included because it is felt that simple programs of this type are available on all of the CFA finalists. The evaluation of tools in this category will therefore be restricted to sort/merge programs and editors (interactive source language editors, interactive object module editors, batch source language editors, and batch object module editors).

(6) Information Retrieval Systems

General purpose application programs operating either on-line (interactively) or in the batch that interpret user requests to locate and display information that is stored either within a structured database or within separate files. These systems can be classified either as query language systems or as report writers.

(4) Layer 1: Operating System Services

Layer 1 implements the operating system services that present a "virtual machine" interface to the services/tools at layers 2 and 3 and manage the real system hardware. The primary functions of the layer 1 tools are listed below:

(1) Memory Management
(2) Processor Management
(3) Timer Management
(4) Interrupt Management
(5) Peripheral Device Management
(6) Input/Output Services
(7) Program Handling Services
(8) Task Management
(9) Instrumentation
(10) Debugging Services
(11) System Startup and Restart
(12) Job Control Facilities
(13) Security
(14) Exception Handling

The layer 1 tool types are generally applicable across all of the software development activities. Layer 1 tool types/capabilities are listed below:

(1) Basic Operating Systems

Runs single user processes from initiation to termination. May or may not overlap I/O with execution. Provides basic I/O support that allows user to refer to files symbolically and to read and write them without knowing the hardware details of the I/O Interface. Provides basic batch supervisor services that control normal and abnormal job termination, job to job transition, and operator communication. Provides a minimum base for program development by supporting at least one language translator and/or linker/loader.

(2) Multiprogramming Operating System

Provides all of the services of the Basic Operating System. Supports the concurrent execution of two or more user jobs by allowing the execution of any job to be suspended while another is executed without any special programming considerations in the user job. Prevents concurrently executing user jobs from accidentally or intentionally destroying each other or the supervisor.

(3) Multiprocessor Operating System

Allows the computing load to be spread across more than one processor based on automatic (programmed) load-leveling algorithms or operator control, but does not require special case programming in the user job. Multiprocessor Operating Systems include the shared storage, loosely coupled, and networked types.

(4) Virtual Machine Monitor

The operating system presents an interface to the user program that makes it appear that the program is executing on a real computing system.

(5) Time-Sharing Operating Systems

This is a variant of the multiprograming operating system in which system resources are allocated to user jobs in such a way that all jobs appear to progress at the same rate. In addition, users are allowed to "interact" with and receive output from their jobs via terminals. Such systems are optimized for response rather than throughput or equipment utilization.

(6) Real-Time Operating Systems

Allows user jobs to be executed within specified short time limits.

(7) Remote Job Entry

A feature of operating systems that allows a batch job, including its input, to be submitted from a device that is connected to the computer system by a teleprocessing line. The device may be a high speed terminal with card and printer devices, a low speed terminal with a keyboard, or another computer.

(8) Checkpoint/Restart

A feature of an operating system that allows a user program to periodically have its state saved so that it may be restarted from a point subsequent to initiation if a hardware or software failure occurs.

4. EVALUATION CRITERIA OF THE SOFTWARE BASES

a. General

One of the fundamental justifications for the entire MCF program is "software capture." That is the capture of the support software investment of the winning architecture. In order to compare the three finalist architectures, a procedure was required which could quantify their support software investment. The following three items were deemed to be identifiable, obtainable and could be translated into financial data. These items are:

a. Applicability
b. Availability by Architecture Manufacturer
c. Availability by other than Architecture Manufacturer

Each of these measures is discussed in the next subsections. Section 4.3 discusses the consolidation of these measures into financial data.

b. Individual Measures

(1) Applicability

This involves determination of the functional relevance of a given software base component (tool type) to the development of military tactical software systems. Applicability is not intended to be a binary criterion but should be a measure of the potential importance of the component, ranging from "not applicable" to "essential." Factors that should be considered in determining the importance of a tool type, in addition to essentiality, include spectrum coverage, economic impact, software size, and the number of different instances of the same tool type for a given CFA. Members were allotted 5,000 points and asked to distribute these points over the tools thereby indicating the relative applicability/importance of each tool. If a tool is essential (in a practical and not a theoretical sense) for military computer software development then it should deserve more weight than one that is only nice to have. Some tools are dependent on others, e.g., a librarian system is heavily dependent on the operating system. Therefore the assignment of a high weight to the librarian systems and the assignment of no weight to the OS would be inconsistent.

Consideration should also be given to spectrum coverage, i.e., the utility of the component across development activities/phases (requirements analysis, software design, etc.) as well as across development disciplines (management, engineering, documentation, validation). Consideration should be given to economic impact, the potential cost savings that may be realized through use of the tool type, or the costs that might be incurred should the component not be available. Software size may be applied to measure the latter. Also, a small tool type (functionally) may deserve less weight than a more powerful tool type and software size may provide some measure of the differences.

Table 3 depicts the ballot which was sent to the committee members. Upon receipt of the ballots the results will be compiled and applied against a predetermined threshold value of 1000 points. All tools which fall below this figure in total points will no longer be considered. The justification for this is that DoD could not afford to build, (in fact it would not be wise to build), tools which a representative spectrum of system developers determined were not applicable in a relative sense.

TABLE 3 BALLOT FOR RELATIVE APPLICABILITY

WEIGHTS

1. LAYER 3

1.1 Requirements Analysis

1.1.1 General Purpose System Simulator ________
1.1.2 System Description Language & Analyzer ________

1.2 Software Design

1.2.1 Computer System Simulator ________
1.2.2 Data Base Design Aid ________
1.2.3 Data Dictionary System ________

1.3 Construct System Tests

1.3.1 Test.Data Generator ________
1.3.2 Test Data Auditor ________
1.3.3 Test Case Design Advisors ________
1.3.3.1 FORTRAN ________
1.3.3.2 COBOL ________
1.3.3.3 CMS-2 (1) ________
1.3.3.4 CMS-2 (2) ________
1.3.3.5 CMS-2 (3) ________
1.3.3.6 JOVIAL J3B ________
1.3.3.7 JOVIAL J73 ________
1.3.3.8 TACPCL ________
1.3.3.9 ATLAS ________
1.3.3.10 OPAL ________
1.3.4 Test Instruments & Analyzers
(BY LANGUAGE) --------

1.4 Build & Unit Test

1.4.1 Assemblers
1.4.1.1 Basic Assembler ________
1.4.1.2 Macro Assembler
1.4.2 Compilers
(BY LANGUAGE) --------
1.4.3 Linkers
1.4.3.1 Basic Linker ________
1.4.3.2 Simple Overlay Linker ________
1.4.3.3 Extended Overlay Linker ________

TABLE 3 BALLOT FOR RELATIVE APPLICABILITY
(Continued)

1.4.4 Debugging Aids

1.4.4.1 Interactive Symbolic Debugger (ASSEMBLER & BY HOL) --------

1.4.4.2 Interactive Absolute Debugger ________

1.4.4.3 Non-Interactive Symbolic Debugger (ASSEMBLER & BY HOL) --------

1.4.4.4 Non-Interactive Absolute Debugger ________

1.4.5 Module Libraries & Change Control Systems ________

1.4.5.1 Basic or Loosely Coupled ________

1.4.5.2 Integrated Library ________

1.4.5.3 Automatic Software Production & Test ________

1.4.6 Performance Monitors

1.4.6.1 Language Dependent Monitors (ASSEMBLER & BY HOL) --------

1.4.6.2 Language Independent Monitor ________

1.4.7 Standards Enforcers (BY LANGUAGE) --------

1.4.8 Preprocessors/Reformatter

1.4.8.1 Preprocessors (BY LANGUAGE) --------

1.4.8.2 Reformatters (BY LANGUAGE) --------

2. LAYER 2

2.1 Data Base Management System ________

2.2 Project Management Aids

2.2.1 PERT/CPM System ________

2.2.2 Project Estimation System ________

2.3 Documentation Aids

2.3.1 Text Processing System ________

2.3.2 Flowchart Construction Language (ASSEMBLER & BY HOL) --------

2.3.3 Automatic Flowcharters (ASSEMBLER & BY HOL) --------

TABLE 3 BALLOT FOR RELATIVE APPLICABILITY
(Continued)

2.4 Data Manipulation Utilities

2.4.1 Sort/Merge ________
2.4.2 Editors ________
2.4.2.1 Interactive Source Language Editors
(ASSEMBLER & BY HOL) --------
2.4.2.2 Interactive Object Module Editors ________
2.4.2.3 Batch Source Language Editors
(ASSEMBLER & BY HOL) --------
2.4.2.4 Batch Object Module Editors ________

2.5 Information Retrieval Systems

2.5.1 Query Language System ________
2.5.2 Report Writer ________

3. LAYER 1

3.1 BOS
3.2 MOS
3.3 MPOS
3.4 TSOS
3.5 TSOS + MPOS
3.6 RTOS
3.7 RTOS + MOS
3.8 RTOS + MPOS
3.9 RTOS + TSOS
3.10 BOS + VMM
3.11 MOS +VMM
3.12 MPOS + VMM
3.13 TSOS + VMM
3.14 TSOS + MPOS + VMM

(2) Availability from Architecture Manufacturers

This criterion involves determining whether the support software tools of Appendix A are available from the finalist architecture manufacturers. A tool will be considered available if it is presently being marketed and maintained. This definition permits the criterion to be applied uniformly and equitably to all the finalist manufacturers. It is felt that, if tools under development were also considered, it would be impossible to determine whether the tool was one month, one year, five years away, etc. from being marketed by the company.

The actual determination of availability of support software tools will be conducted in two phases. Phase I will consist of providing each of the manufacturers a list of the support software tools (Table 3) as well as the descriptions of Appendix A and requesting them to answer in the affirmative for all tools which they actively market and maintain. Phase II will consist of a visit to each of the manufacturers by government personnel and an independent auditor to obtain supporting documentation and to audit the manufacturer responses.

At the end of Phase II, an accurate list of available support software for each manufacturer will be delineated.

(3) Availability From Other Vendors

It was felt that there exists a great deal of support software available from sources other than the architecture manufacturers which meets the technical descriptions of Appendix A. Such software should be included. Again, a firm criterion is needed and the following was selected: The tool must meet the specific requirements of the applicable part of Appendix A, must be hosted on and targeted for one of the finalist architectures, and must be marketed and maintained. The International Computer Programs Inc. (ICP) "INTERFACE Reference Series" has been chosen as the source document because it has become a defacto standard for software marketing. However only tools which were not marketed by the manufacturer were searched for. Upon finding such a tool in the ICP document, the vendor was contacted to obtain further supporting data.

c. Consolidation of Individual Measures

The desired end product of the entire software base evaluation is a dollar figure for each architecture's existing software base, as well as a schedule depicting the cost of and how the deficiencies in the software base can be eliminated. From the measures discussed in Section 4.2, lists of available software tools were generated for each architecture. Also, for each architecture, a list of software deficiencies were generated in development sequence. The development sequence were based upon applicability and PERT ordering. In other words if tools X, Y, and Z are rated by members in terms of applicability in the order Z, Y, X, but, the development of Y before Z would incur a saving in overall development costs, then the final development sequence would be ordered as Y, Z, X.

Utilizing these figures, an overall figure for the software base of each architecture was generated. Also, assuming a figure of 2 million dollars a year as an estimate of the support software R&D dollars available when the MCF program is implemented, and utilizing the deficiency lists in development order, a development schedule was generated for each architecture which provides, at a glance, the relative future deficiencies of each architecture.

5. RESULTS OF THE EVALUATION OF THE SOFTWARE BASES

a. Results by Individual Measures

(1) Applicability

Early in May 1976, the Software Evaluation Methodology Subcommittee forwarded to all voting members of the Architecture Selection Committee the ballot depicted in Table 3 of the previous section. Each member was requested to vote on the applicability of the tools delineated in that Table by distributing a maximum of 5,000 points over the tools.

Table 4 contains the results of the balloting. There were a few problems with the balloting, the most notable of which was that more than half of the members failed to vote on particular compilers, but cast their vote for compilers in general. Therefore, it was decided that compilers for each of the DoD approved languages as follows should be included in the list of applicable tools:

FORTRAN
COBOL
CMS-2 (1)
CMS-2 (2)
CMS-2 (3)
JOVIAL J3B
JOVIAL J73
TACPOL

Automatic test equipment languages (ATLAS and OPAL) were removed from consideration since none were available for the finalist architectures.

As can be seen, the threshold of 1000 points divides the list approximately in half leaving 28 tools to be utilized in the availability phase of the evaluation.

(2) Availability from Architectures Manufacturers

In order to determine the availability of support software tools, two steps were taken. First, Table 3, which lists all of the support software tools of interest to this evaluation, was forwarded to the three finalist architecture manufacturers along with the Software Evaluation Methodology Subcommittee's Report entitled "Procedure for Ranking the Software Bases of Candidate Architectures for the Military Computer Family." Each manufacturer was asked to read the report, especially the description of the minimum characteristics of the tools and indicate which of the Table 3 tools are marketed and maintained by the company. Table 5 contains the results of that process.

Next a meeting was set up by the chairman of the Software Evaluation Methodology Subcommittee with each of the three manufacturers. Dr. H. Stone, the independent auditor, and a representative of the subcommittee chairman conducted these meetings and requested that each manufacturer supply supporting documentation to substantiate their claims of tool existence/marketing. Also, each manufacturer was requested to provide an estimate of the number of source lines of code, language utilized, and object code size in instructions for each tool.

TABLE 4

COMPILED RESULTS OF THE APPLICABILITY BALLOTING

9708	Compilers
4405	Macro Assemblers
4317	Interactive Source Language Editors
3752	Interactive Symbolic Debuggers
3367	Extended Overlay Linker
3010	Test Case Design Advisors
2969	Integrated Library
2687	Text Processing System
2677	DBMS
2380	GP System Simulator
2270	TSOS + VMM
1872	Language Independent Monitors
1688	Test Data Generator
1645	Non-Interactive Symbolic Debugger
1623	Computer System Simulator
1535	Batch Source Language Editors
1418	Language Dependent Monitors
1400	TSOS + MPOS + VMM
1390	Basic Assembler
1310	RTOS + TSOS
1217	Test Instrumenters & Analyzers
1187	Automatic SW Production & Test
1158	Basic Linker
1140	Standards Enforcers
1110	Reformatters
1095	Test Data Auditor
1008	Simple Overlay Linker
1006	Data Base Design Aid

975	Sort/Merge
880	Preprocessors
855	Basic or Loosely Coupled Library
834	Project Estimation System
808	System Description Language & Analyzer
795	RTOS
792	PERT/CPM
760	TSOS
757	MOS
705	BOS
675	Report Writers
658	Interactive Object Module Editors

TABLE 4

COMPILED RESULTS OF THE APPLICABILITY BALLOTING
(Continued)

642	Interactive Absolute Debugger
622	Data Dictionary System
615	RTOS + MPOS
607	Query Language Systems
600	PDL
600	Batch Object Module Editors
600	Automatic Flowcharters
558	Flowchart Construction Languages
513	Non-Interactive Absolute Debugger
410	MPOS
365	TSOS + MPOS
350	BOS + VMM
315	MOS + VMM
305	RTOS + MOS
250	MPOS + VMM

TABLE 5 SOFTWARE AVAILABILITY AS CITED BY THE MANUFACTURERS

	IBM	DEC	INTERDATA
1. LAYER 3			
1.1 Requirements Analysis			
1.1.1 General Purpose System Simulator	Y	N	N
1.2 Software Design			
1.2.1 Computer System Simulator	N	N	N
1.2.2 Data Base Design Aid	Y	Y	N
1.3 Construct System Tests			
1.3.1 Test Data Generator	Y	N	N
1.3.2 Test Data Auditor	N	N	N
1.3.3 Test Case Design Advisors			
1.3.3.1 FORTRAN	N	N	Y
1.3.3.2 COBOL	N	N	N
1.3.3.3 CMS-2 (1)	N	N	N
1.3.3.4 CMS-2 (2)	N	N	N
1.3.3.5 CMS-2 (3)	N	N	N
1.3.3.6 JOVIAL J3B	N	N	N
1.3.3.7 JOVIAL J73	N	N	N
1.3.3.8 TACPOL	N	N	N
1.3.4 Test Instruments & Analyzers (BY LANGUAGE)	N	N	N
1.4 Build & Unit Test			
1.4.1 Assemblers			
1.4.1.1 Basic Assembler	Y	Y	Y
1.4.1.2 Macro Assembler	Y	Y	Y
1.4.2 Compilers			
1.4.2.1 FORTRAN	Y	Y	Y
1.4.2.2 COBOL	Y	Y	Y
1.4.2.3 CMS-2 (1)	N	N	N
1.4.2.4 CMS-2 (2)	N	N	N
1.4.2.5 CMS-2 (3)	N	N	N
1.4.2.6 JOVIAL J3B	N	N	N
1.4.2.7 JOVIAL J73	N	N	N
1.4.2.8 TACPOL	N	N	N

TABLE 5 SOFTWARE AVAILABILITY AS CITED BY THE MANUFACTURERS
(Continued)

1.4.3	Linkers			
1.4.3.1	Basic Linker	Y	Y	Y
1.4.3.2	Simple Overlay Linker	N	Y	Y
1.4.3.3	Extended Overlay Linker	Y	Y	N
1.4.4	Debugging Aids			
1.4.4.1	Interactive Symbolic Debugger (ASSEMBLER & BY HOL)	Y	N	N
1.4.4.3	Non-Interactive Symbolic Debugger (ASSEMBLER & BY HOL)	Y	N	N
1.4.5	Module Libraries & Change Control Systems			
1.4.5.2	Integrated Library	N	N	Y
1.4.5.3	Automatic Software Production & Test	N	N	Y
1.4.6	Performance Monitors			
1.4.6.1	Language Dependent Monitors (ASSEMBLER & BY HOL)	N	N	N
1.4.6.2	Language Dependent Monitor	N	N	N
1.4.7	Standards Enforcers (BY LANGUAGE)	N	N	N
1.4.8	Preprocessors/Reformatter			
1.4.8.2	Reformatters (BY LANGUAGE)	N	N	N
2.	LAYER 2			
2.1	Data Base Management System	Y	Y	N
2.3	Documentation Aids			
2.3.1	Text Processing System	Y	Y	Y
2.4	Data Manipulation Utilities			
2.4.2	Editors			
2.4.2.1	Interactive Source Language Editors (ASSEMBLER & BY HOL)	Y	Y	Y
2.4.2.3	Batch Source Language Editors (ASSEMBLER & BY HOL)	Y	Y	Y
3.	LAYER 1			
3.9	RTOS + TSOS	Y	Y	N
3.13	TSOS + VMM	Y	N	N
3.14	TSOS + MPOS + VMM	Y	N	N

Dr. H. Stone has documented this audit which is included as Appendix B to this report. Table 6 contains the result of the audit.

(3) Availability from Other Vendors

Software tools which meet the technical requirements of Section 3.4 and were marketed by other than the manufacturers also were investigated. The International Computer Programming Inc. (ICP) "INTERFACE Reference Series," Jan 1976, was utilized as the source document for this investigation. Tables 7-9 list the support software tools by tool type. Also listed is the marketing company as well as the one installation sale or yearly lease cost. Each of the companies was contacted and requested to provide further detailed documentation on the product. As stated earlier, if an architecture manufacturer marketed a particular tool, that tool was not sought in the ICP document. Table 10 depicts an overall composite of software availability from sources other than the architecture manufacturer. The dashes indicate tools which the manufacturer did market and therefore were not looked for in the ICP document.

b. Consolidation of Results

The main purpose of the Software Evaluation was to determine the relative current software dollar investment of the three architectures. Also it was desired to obtain the relative deficiencies of the architectures in terms of dollars. In order to do this, a development cost for each tool was needed. It was known that the software vendors considered such information to be proprietary. Therefore, the following approach was taken: First each manufacturer was requested to provide the source code size for his available tools as well as the language which the tool was written in, and the object code size in instructions. Second, a productivity figure was needed. F. Brooks', "The Mythical Man-Month" was considered to be the best source since it compiled productivity figures from the IBM's OS/360 development as well as Bell Labs' ESS software development. Brooks cites 600 lines of code per man-year for operating system development and 2,000 lines of code per man-year for other support software development. It is felt that the state-of-the-art in operating system has improved significantly since his data was obtained (nearly ten years ago) and thus a figure of 1,000 and 2,000 lines of code per man-year for operating system and other support software, respectively, was decided upon.

It should be noted that anticipated maintenance costs of support software have not been estimated in this analysis. A question could be raised as to accuracy of the analysis without this consideration. However, it was felt that it would be impossible to estimate accurate costs for the maintenance of the support software tools because many of these tools developed under the MCF program may be adopted, marketed and maintained by the selected architecture manufacturer or by the developer of the tools.

Table 11 depicts the source lines of code as supplied by the manufacturer. As can be seen, this data is scarce and therefore estimates had to be made of the cost of tools for which there was no data. For all tools, a figure of $70,000 a man-year was utilized.

TABLE 6 SOFTWARE AVAILABILITY AS DETERMINED BY AUDIT OF MANUFACTURERS RESPONSES

	IBM	Dec	Interdata
1. LAYER 3			
1.1 Requirements Analysis			
1.1.1 General Purpose System Simulator	Y	N	N
1.2 Software Design			
1.2.1 Computer System Simulator	N	N	N
1.2.2 Data Base Design Aid	Y	Y	N
1.3 Construct System Tests			
1.3.1 Test Data Generator	N	N	N
1.3.2 Test Data Auditor	N	N	N
1.3.3 Test Case Design Advisors			
1.3.3.1 FORTRAN	N	N	N
1.3.3.2 COBOL	N	N	N
1.3.3.3 CMS-2	N	N	N
1.3.3.4 JOVIAL	N	N	N
1.3.3.5 TACPOL	N	N	N
1.3.4 Test Instruments & Analyzers			
1.3.4.1 FORTRAN	Y	N	N
1.3.4.2 COBOL	Y	N	N
1.3.4.3 CMS-2	N	N	N
1.3.4.4 JOVIAL	N	N	N
1.3.4.5 TACPOL	N	N	N
1.4 Build & Unit Test			
1.4.1 Assemblers			
1.4.1.1 Basic Assembler	Y	Y	Y
1.4.1.2 Macro Assembler	Y	Y	Y
1.4.2 Compilers			
1.4.2.1 FORTRAN	Y	Y	Y
1.4.2.2 COBOL	Y	Y	Y
1.4.2.3 CMS-2	N	N	N
1.4.2.7 JOVIAL	Y	N	N
1.4.2.8 TACPOL	N	N	N
1.4.3 Linkers			
1.4.3.1 Basic Linker	Y	Y	Y
1.4.3.2 Simple Overlay Linker	Y	Y	Y
1.4.3.3 Extended Overlay Linker	Y	Y	N

TABLE 6 SOFTWARE AVAILABILITY AS DETERMINED BY AUDIT OF MANUFACTURERS RESPONSES (Continued)

1.4.4	Debugging Aids			
1.4.4.1	Interactive Symbolic Debugger			
1.4.4.1.1	Assembler	N	N	N
1.4.4.1.2	FORTRAN	Y	N	N
1.4.4.1.3	COBOL	Y	N	N
1.4.4.1.4	CMS-2	N	N	N
1.4.4.1.5	JOVIAL	N	N	N
1.4.4.1.6	TACPOL	N	N	N
1.4.4.3	Non-Interactive Symbolic Debuggers			
1.4.4.3.1	Assembler	N	N	N
1.4.4.3.2	FORTRAN	Y	Y	N
1.4.4.3.3	COBOL	Y	Y	N
1.4.4.3.4	CMS-2	N	N	N
1.4.4.3.5	JOVIAL	N	N	N
1.4.4.3.6	TACPOL	N	N	N
1.4.5	Module Libraries & Change Control Systems			
1.4.5.2	Integrated Library	N	N	N
1.4.5.3	Automatic Software Production & Test	N	N	N
1.4.6	Performance Monitors			
1.4.6.1	Language Dependent Monitor			
1.4.6.1.1	Assembler	N	N	N
1.4.6.1.2	FORTRAN	Y	N	N
1.4.6.1.3	COBOL	Y	N	N
1.4.6.1.4	CMS-2	N	N	N
1.4.6.1.5	JOVIAL	N	N	N
1.4.6.1.6	TACPOL	N	N	N
1.4.6.2	Language Independent Monitor	N	N	N
1.4.7	Standards Enforcers			
1.4.7.1	FORTRAN	N	N	N
1.4.7.2	COBOL	N	N	N
1.4.7.3	CMS-2	N	N	N
1.4.7.4	JOVIAL	N	N	N
1.4.7.5	TACPOL	N	N	N
1.4.8	Preprocessors, Reformatters			
1.4.8.2	Reformatters			
1.4.8.2.1	FORTRAN	N	N	N
1.4.8.2.2	COBOL	N	N	N
1.4.8.2.3	CMS-2	N	N	N
1.4.8.2.4	JOVIAL	N	N	N
1.4.8.2.5	TACPOL	N	N	N

TABLE 6 SOFTWARE AVAILABILITY AS DETERMINED BY AUDIT OF MANUFACTURERS RESPONSES (Continued)

2. LAYER 2			
2.1 Data Base Management System	Y	Y	N
2.3 Documentation Aids			
2.3.1 Text Processing System	Y	Y	N
2.4 Data Manipulation Utilities			
2.4.2 Editors			
2.4.2.1 Interactive Source Language Editors	Y	Y	Y
2.4.2.3 Batch Source Language Editors	Y	Y	Y
3. LAYER 1			
3.9 RTOS + TSOS	Y	Y	Y
3.13 TSOS + VMM	Y	N	N
3.14 TSOS + MPOS + VMM	Y	N	N

TABLE 7 - SOFTWARE MARKETED
BY
OTHER THAN IBM
FOR THE
IBM 360/370 ARCHITECTURE

1.2.1	Computer System Simulator	
a.	SCERT-76 Marketed by COMTEN, Inc., Rockville, MD	$22,000
b.	CASE Marketed by TESTDATA Systems Corporation, McLean, VA	$17,000
1.3.1	Test Data Generator	
a.	DATAMACS Marketed by Management & Computer Services, Inc. Valley Forge, PA	$ 8,500
b.	TDG-L Test Data Generating Language Marketed by K&A Software Products, Dallax, TX	$11,000
1.3.2	Test Data Auditor	
a.	APRO/Test File Checker Marketed by Synergistics Corporation, Burlington, PA	$ 3,800
1.4.4.1.1	Interactive Symbolic Debugger (Assembler)	
a.	SYMBUG-A Marketed by Standard Data Corporation, NY, NY	$17,500
1.4.5.2	Integrated Library	
a.	PULMACS III Marketed by Management & Computer Services, Inc., Valley Forge, PA	$ 3,000
b.	SUR (Software Utility Routine) Marketed by Boeing Computer Services, Inc., Seattle, WA	$ 5,000
c.	The LIBRARIAN Marketed by Applied Data Research, Inc., Princeton, NJ	$ 5,300
d.	PANVALET Marketed by Pansophic Systems, Inc. Oakbrook, IL	$ 5,580

TABLE 7 - SOFTWARE MARKETED
BY
OTHER THAN IBM
IBM 360/370 ARCHITECTURE
(Continued)

1.4.6.1.1 Language Dependent Monitor (Assembly)

a. STROBE
Marketed by Programart Corporation, Cambridge, MA $ 9,400

1.4.6.2 Language Independent Monitor

a. Problem Program Evaluator - PPE
Marketed by Broole & Baggage, Sunnyvale, CA $ 9,800

b. Operating System Monitor
Marketed by Lincoln Land Software Systems, Inc.,
Springfield, IL $2,400/first installation
1,200/each additional installation

1.4.8.2.1 Reformatter (FORTRAN)

a. IFTRAN-2 FORTRAN Preprocessor for Structured Programming
Marketed by General Research Corporation,
Santa Barbara, CA $ 1,000

1.4.8.2.2 Reformatter (COBOL)

a. Reformat-Automatic Standardization of COBOL Source Programs, Plus Shorthand COBOL
Marketed by EDP Management, Inc., LaMesa, CA $ 3,500

TABLE 8 - SOFTWARE MARKETED BY OTHER THAN DEC FOR THE PDP-11 ARCHITECTURE

1.3.4.1 Test Instrumenter & Analyzer (FORTRAN)

a. RXVP-1 Automated Verification System for FORTRAN
Marketed by General Research Corporation
Santa Barbara, CA $14,000

b. TAP Testing Analysis Package
Marketed by General Research Corporation
Santa Barbara, CA $ 7,500 lease only

1.4.4.1.3 Interactive Symbolic Debugger (COBOL)

a. Dependable Debugging Tool
Marketed by Innovative Systems Association
New York, NY $225.00

1.4.8.2.1 Reformatter (FORTRAN)

a. IFTRAN-2 FORTRAN Preprocessor for Structured Programming
Marketed by General Research Corporation
Santa Barbara, CA $1,000

TABLE 9 - SOFTWARE MARKETED BY OTHER THAN INTERDATA FOR THE INTERDATA 8/32 ARCHITECTURE

1.3.4.1 Test Instrumenter & Analyzer (FORTRAN)

a. RXVP-1 Automated Verification System for FORTRAN
Marketed by General Research Corporation
Santa Barbara, CA $14,000

b. TAP - Testing Analysis Package
Marketed by General Research Corporation
Santa Barbara, CA $ 7,500/lease only

1.4.8.2.1 Reformatter (FORTRAN)

a. IFTRAN-2 FORTRAN Preprocessor for Structured Programming
Marketed by General Research Corporation
Santa Barbara, CA $ 1,000

TABLE 10 COMPOSITE OF SOFTWARE AVAILABILITY FOR THE IBM, DEC AND INTERDATA ARCHITECTURES FROM OTHER SOURCES THAN IBM, DEC AND INTERDATA

	IBM	DEC	INTERDATA
1. LAYER 3			
1.1 Requirements Analysis			
1.1.1 General Purpose System Simulator	-	N	N
1.2 Software Design			
1.2.1 Computer System Simulator	Y	N	N
1.2.2 Data Base Design Aid	-	-	N
1.3 Construct System Tests			
1.3.1 Test Data Generator	Y	N	N
1.3.2 Test Data Auditor	Y	N	N
1.3.3 Test Case Design Advisors			
1.3.3.1 FORTRAN	N	N	N
1.3.3.2 COBOL	N	N	N
1.3.3.3 CMS-2	N	N	N
1.3.3.4 JOVIAL	N	N	N
1.3.3.5 TACPOL	N	N	N
1.3.4 Test Instruments & Analyzers (BY LANGUAGE)			
1.3.4.1 FORTRAN	-	Y	Y
1.3.4.2 COBOL	-	N	N
1.3.4.3 CMS-2	N	N	N
1.3.4.4 JOVIAL	N	N	N
1.3.4.5 TACPOL	N	N	N
1.4 Build & Unit Test			
1.4.1 Assemblers			
1.4.1.1 Basic Assembler	-	-	-
1.4.1.2 Macro Assembler	-	-	-
1.4.2 Compilers			
1.4.2.1 FORTRAN	-	-	-
1.4.2.2 COBOL	-	-	-
1.4.2.3 CMS-2	N	N	N
1.4.2.4 JOVIAL	-	N	N
1.4.2.8 TACPOL	N	N	N

TABLE 10 COMPOSITE OF SOFTWARE AVAILABILITY FOR THE IBM, DEC AND INTERDATA ARCHITECTURES FROM OTHER SOURCES THAN IBM, DEC AND INTERDATA (CONTINUED)

		IBM	DEC	INTERDATA
1.4.3	Linkers			
1.4.3.1	Basic Linker	-	-	-
1.4.3.2	Simple Overlay Linker	-	-	-
1.4.3.3	Extended Overlay Linker	-	-	N
1.4.4	Debugging Aids			
1.4.4.1	Interactive Symbolic Debugger			
1.4.4.1.1	Assembly	Y	N	N
1.4.4.1.2	FORTRAN	-	N	N
1.4.4.1.3	COBOL	-	Y	N
1.4.4.1.4	CMS-2	N	N	N
1.4.4.1.5	JOVIAL	N	N	N
1.4.4.1.6	TACPOL	N	N	N
1.4.4.3	Non-interactive Symbolic Debugger			
1.4.4.3.1	Assembler	N	N	N
1.4.4.3.2	FORTRAN	-	-	N
1.4.4.3.3	COBOL	-	-	N
1.4.4.3.4	CMS-2	N	N	N
1.4.4.3.5	JOVIAL	N	N	N
1.4.4.3.6	TACPOL	N	N	N
1.4.5	Module Libraries & Change Control Systems			
1.4.5.2	Integrated Library	Y	N	N
1.4.5.3	Automatic Software Production & Test	N	N	N
1.4.6	Performance Monitors			
1.4.6.1	Language Dependent Monitors			
1.4.6.1.1	Assembly	Y	N	N
1.4.6.1.2	FORTRAN	-	N	N
1.4.6.1.3	COBOL	-	N	N
1.4.6.1.4	CMS-2	N	N	N
1.4.6.1.5	JOVIAL	N	N	N
1.4.6.1.6	TACPOL	N	N	N
1.4.6.2	Language Independent Monitor	Y	N	N
1.4.7	Standards Enforcers			
1.4.7.1	FORTRAN	N	N	N
1.4.7.2	COBOL	N	N	N
1.4.7.3	CMS-2	N	N	N
1.4.7.4	JOVIAL	N	N	N
1.4.7.5	TACPOL	N	N	N
1.4.8	Pre-processors/Reformatter			
1.4.8.2	Reformatter			

TABLE 10 COMPOSITE OF SOFTWARE AVAILABILITY FOR THE IBM, DEC AND INTERDATA ARCHITECTURES FROM OTHER SOURCES THAN IBM, DEC AND INTERDATA (CONTINUED)

	IBM	DEC	INTERDATA
1.4.8.2.1 FORTRAN	Y	Y	Y
1.4.8.2.2. COBOL	Y	N	N
1.4.8.2.3 CMS-2	N	N	N
1.4.8.2.4 JOVIAL	N	N	N
1.4.8.2.5 TACPOL	N	N	N
2. LAYER 2			
2.1 Data Base Management System	-	-	N
2.3 Documentation Aids			
2.3.1 Text Processing System	-	-	N
2.4 Data Manipulation Utilities			
2.4.2 Editors			
2.4.2.1 Interactive Source Language Editors	-	-	-
2.4.2.3 Batch Source Language Editors	-	-	-
3. LAYER 1			
3.9 RTOS + TSOS	-	-	-
3.13 TSOS + VMM	-	N	N
3.14 TSOS + MPOS + VMM	-	N	N

TABLE 11 - ESTIMATES OF TOOLS SIZE
BY
SOURCE LINES OF CODE
AND
ESTIMATED TOOL COST

		IBM (Source Lines)	DEC (Source Lines)	Interdata (Source Lines)	Gov't Estimate ($)	Combined Estimate ($)
1.1.1	General Purpose System Simulator	20,000 Assembly				700K
1.2.1	Computer System Simulator				420K	420K
1.2.2	Data Base Design Aid				1150K	1150K
1.3.1	Test Data Generator				350K	350K
1.3.2	Test Data Auditor				140K	140K
1.3.3	Test Case Design Advisors				2100K (for all languages)	2100K (for all languages)
	1.3.3.1 FORTRAN					
	1.3.3.2 COBOL					
	1.3.3.3 CMS-2					
	1.3.3.4 JOVIAL					
	1.3.3.5 TACPOL					
1.3.4	Test Instruments & Analyzers				1400K (for all languages)	1400K (for all languages)
	1.3.4.1 FORTRAN					
	1.3.4.2 COBOL					
	1.3.4.3 JOVIAL					
	1.3.4.5 TACPOL					
1.4.1.1	Basic Assembler				280K	280K
1.4.1.2	Macro Assembler	40,000 Assembly				1400K
1.4.2	Compilers					
	1.4.2.1 FORTRAN	75,000 FORTRAN & Assembly				2600K

TABLE 11 - ESTIMATES OF TOOLS SIZE
BY
SOURCE LINES OF CODE
AND
ESTIMATED TOOL COST

	IBM (Source Lines)	TEC (Source Lines)	Interdata (Source Lines)	Gov't Estimate ($)	Updated Estimate ($)
1.4.2.1 COBOL	300,000 Assembly				10,500K
1.4.2.3 CMS-2				4900K for JOVIAL, CMS-2 & TACPOL	4900K for JOVIAL, CMS-2 & TACPOL
1.4.2.4 JOVIAL					
1.4.2.5 TACPOL				3500K for CMS-2 & TACPOL	3500K for CMS-2 & TACPOL
1.4.3.1 Basic Linker				210K	210K
1.4.3.2 Simple Overlay Linker				210K	210K
1.4.3.3 Extended Overlay Linker	15,000 Assembly				530K
1.4.4.1 Interactive Symbolic Debugger				1400K (for all languages)	1400K (for all languages)
1.4.4.1.1 ASSEMBLY					
1.4.4.1.2 FORTRAN	15,000 HOL				
1.4.4.1.3 COBOL					
1.4.4.1.4 CMS-2					
1.4.4.1.5 JOVIAL					
1.4.4.1.6 TACPOL					
1.4.4.3 Non-Interactive Symbolic Debugger				280K (for all languages)	280K (for all languages)
1.4.4.3.1 ASSEMBLY					
1.4.4.3.2 FORTRAN					
1.4.4.3.3 COBOL					
1.4.4.3.4 CMS-2					
1.4.4.3.5 JOVIAL					
1.4.4.3.6 TACPOL					

TABLE 11 - ESTIMATES OF TOOLS SIZE
BY
SOURCE LINES OF CODE
AND
ESTIMATED TOOL COST

		IBM (Source Lines)	DEC (Source Lines)	Interdata (Source Lines)	Gov't Estimate ($)	Combined Estimate ($)
1.4.5.2	Integrated Library				700K	700K
1.4.5.3	Automatic Software Production & Test				1000K	1000K
1.4.6.1	Language Dependent Monitors				1050K (for all languages)	1050K (for all languages)
	1.4.6.1.1 ASSEMBLY					
	1.4.6.1.2 FORTRAN					
	1.4.6.1.3 COBOL					
	1.4.6.1.4 CMS-2					
	1.4.6.1.5 JOVIAL					
	1.4.6.1.6 TACPOL					
1.4.6.2	Language Independent Monitor				210K	210K
1.4.7	Standard Enforcers				420K (for all languages)	420K (for all languages)
	1.4.7.1 FORTRAN					
	1.4.7.2 COBOL					
	1.4.7.3 CMS-2					
	1.4.7.4 JOVIAL					
	1.4.7.5 TACPOL					
1.4.8	Reformatters				560K (for all languages)	560K (for all languages)
	1.4.8.1 FORTRAN					
	1.4.8.2 COBOL					
	1.4.8.3 CMS-2					
	1.4.8.4 JOVIAL					
	1.4.8.5 TACPOL					
2.1	Data Base Management System	119K Assembly Language 32K Run-time Words				42,000K

TABLE 11 - ESTIMATES OF TOOLS SIZE
BY
SOURCE LINES OF CODE
AND
ESTIMATED TOOL COST

		IBM (Source Lines)	DEC (Source Lines)	Interdata (Source Lines)	Gov't Estimate ($)	Contract Estimate ($)
2.3.1	Text Processing System				630K	630K
2.4.2.1	Interactive Source Language Editor				560K	560K
2.4.2.3	Batch Source Language Editor				210K	210K
3.9	RTOS + TSOS	48K Assembly Language				3500K
3.13	TSOS + VMM				2800K	2800K
3.14	TSOS + MPOS + VMM				1400K	1400K

Tables 12 - 14 depict the software bases for the IBM 360/370, DEC PDP-11 and the Interdata 8/32. Tables 15 - 17 list the deficiencies of each architecture and proposed development sequence as well as the estimated cost of developing each tool.

Tables 18 - 20 depict schedules for software base deficiency correction of the three architectures based upon an estimated expenditure of $2,000,000 a year, while Tables 21 - 23 depict similar schedules based upon $3,000,000 a year and Tables 24 - 26 depict schedules based upon $1,000,000 a year.

TABLE 12 - IBM 360/370 SOFTWARE BASE

		Estimated Cost
1.	General Purpose System Simulator	700K
2.	Computer System Simulator	420K
3.	Data Base Design Aid	1150K
4.	Test Data Generator	350K
5.	Test Data Auditor	140K
6.	Test Insturmenter & Analyzer (FORTRAN, COBOL)	500K
7.	Basic Assembler	280K
8.	Macro Assembler	1400K
9.	Compiler (FORTRAN, COBOL, JOVIAL)	11,600
10.	Basic Linker	210K
11.	Simple Overlay Linker	210K
12.	Extended Overlay Linker	560K
13.	Interactive Symbolic Debugger (ASSEMBLER, FORTRAN, COBOL)	700K
14.	Non-Interactive Symbolic debugger (FORTRAN, COBOL)	90K
15.	Integrated Library	700K
16.	Language Dependent Monitor (ASSEMBLER, FORTRAN, COBOL)	525K
17.	Language Independent Monitor	210K
18.	Reformatter (FORTRAN, COBOL)	224K

TABLE 12 - IBM 360/370 SOFTWARE BASE
(Continued)

		Estimated Cost
19.	Data Base Management System	4200K
20.	Text Processing System	630K
21.	Interactive Source Language Editors	560K
22.	Batch Source Language Editors	210K
23.	RTOS + TSOS	2500K
24.	TSOS + VMM	2800K
25.	TSOS + MPOS + VMM	1400K
	TOTAL	32,269K

TABLE 13 - DEC PDP-11 SOFTWARE BASE

		Estimated Cost
1.	Data Base Design Aid	1150K
2.	Test Insturmenter & Analyzer (FORTRAN)	280K
3.	Basic Assembler	280K
4.	Macro Assembler	1400K
5.	Compiler (FORTRAN, COBOL)	9600K
6.	Basic Linker	210K
7.	SIMRC Overlay Linker	210K
8.	Extended Overlay Linker	560K
9.	Interactive Symbolic Debugger (COBOL)	230K
10.	Non-Interactive Symbolic Debugger (FORTRAN, COBOL)	90K
11.	Reformatter (FORTRAN)	110K
12.	Data Base Management System	4200K
13.	Text Processing System	630K
14.	Interactive Source Language Editor	560K
15.	Batch Source Language Editor	210K
16.	RTOS + TSOS	2500K
	TOTAL	22.220K

TABLE 14 - INTERDATA 8/32 SOFTWARE BASE

		Estimated Cost
1.	Test Instrumenters & Analyzers (FORTRAN)	280K
2.	Basic Assembler	280K
3.	Macro Assembler	1400K
4.	Compilers (FORTRAN, COBOL)	9600K
5.	Basic Linker	210K
6.	Simple Overlay Linker	210K
7.	Reformatter (FORTRAN)	110K
8.	Interactive Source Language Editor	560K
9.	Batch Source Language Editor	210K
10.	RTOS + TSOS	2500K
	TOTAL	15,360K

TABLE 15 - COMPOSITE SOFTWARE DEFICIENCIES
OF
IBM 360/370 ARCHITECTURE
(ORDERED IN PROPOSED DEVELOPMENT SEQUENCE)

		Estimated Cost
1.	Compilers - CMS-2, TACPOL	3500K
2.	Interactive Symbolic Debugger - CMS-2, JOVIAL, TACPOL	700K
3.	Non-Interactive Symbolic Debugger - ASSEMBLER CMS-2 JOVIAL TACPOL	180K
4.	Test Case Design Advisor - FORTRAN COBOL CMS-2 JOVIAL TACPOL	2100K
5.	Language Dependent Monitors - CMS-2 JOVIAL TACPOL	525K
6.	Standards Enforcers - FORTRAN COBOL CMS-2 JOVIAL TACPOL	420K
7.	Reformatters - CMS-2 JOVIAL TACPOL	330K
8.	Test Instrumenters & Analyzers - CMS-2 JOVIAL TACPOL	840K
9.	Automatic Software Production & Test	1000K
	TOTAL	9,595K

TABLE 16 - COMPOSITE SOFTWARE DEFICIENCIES OF DEC PDP-11 ARCHITECTURE (ORDERED IN PROPOSED DEVELOPMENT SEQUENCE)

			Estimated Cost
1.	Compilers -	CMS-2 JOVIAL TACPOL	4900K
2.	Integrated Library		700K
3.	Interactive Symbolic Debugger -	ASSEMBLER FORTRAN CMS-2 JOVIAL TACPOL	1200K
4.	Non-Interactive Symbolic Debugger -	ASSEMBLER CMS-2 JOVIAL TACPOL	180K
5.	Test Case Design Advisor -	FORTRAN COBOL CMS-2 JOVIAL TACPOL	2100K
6.	Test Data Generator		350K
7.	Language Independent Monitor		210K
8.	Language Dependent Monitor -	ASSEMBLER FORTRAN COBOL CMS-2 JOVIAL TACPOL	1050K
9.	General Purpose System Simulator		700K
10.	Computer System Simulator		420K

TABLE 16 - COMPOSITE SOFTWARE DEFICIENCIES OF DEC PDP-11 ARCHITECTURE (ORDERED IN PROPOSED DEVELOPMENT SEQUENCE) (Continued)

		Estimated Cost
11.	Standards Enforcers - FORTRAN COBOL CMS-2 JOVIAL TACPOL	420K
12.	Test Data Auditor	140K
13.	Reformatters - COBOL CMS-2 JOVIAL TACPOL	440K
14.	Test Instrumenters & Analyzers - COBOL CMS-2 JOVIAL TACPOL	1120K
15.	Automatic Software Production & Test	1000K
16.	TSOS + VMM	2800K
17.	TSOS + MPOS + VMM	1400K
	TOTAL	19,130K

TABLE 17 - COMPOSITE SOFTWARE DEFICIENCIES
OF
INTERDATA 8/32 ARCHITECTURE
(ORDERED IN PROPOSED DEVELOPMENT SEQUENCE) .

			Estimated Cost
1.	Compilers -	CMS-2 JOVIAL TACPOL	4900K
2.	Integrated Library		700K
3.	Entended Overlay Linker		560K
4.	Interactive Symbolic Debugger -	ASSEMBLER FORTRAN COBOL CMS-2 JOVIAL TACPOL	1400K
5.	Non-Interactive Symbolic Debugger -	ASSEMBLER FORTRAN COBOL CMS-2 JOVIAL TACPOL	280K
6.	Test Case Design Advisor -	FORTRAN COBOL CMS-2 JOVIAL TACPOL	2100K
7.	Text Processing System		630K
8.	Test Data Generator		350K
9.	Language Independent Monitor		210K
10.	Language Dependent Monitor -	ASSEMBLER FORTRAN COBOL CMS-2 JOVIAL TACPOL	1050K

TABLE 17 - COMPOSITE SOFTWARE DEFICIENCIES OF INTERDATA 8/32 ARCHITECTURE (ORDERED IN PROPOSED DEVELOPMENT SEQUENCE) (Continued)

		Estimated Cost
11.	Data Base Management System	4200K
12.	Data Base Design Aid	1150K
13.	General Purpose System Simulator	700K
14.	Computer System Simulator	420K
15.	Standards Enforcers - FORTRAN COBOL CMS-2 JOVIAL TACPOL	420K
16.	Test Data Auditor	140K
17.	Reformatters - COBOL CMS-2 JOVIAL TACPOL	440K
18.	Test Instrumenter & Analyzer - COBOL CMS-2 JOVIAL TACPOL	1120K
19.	Automatic Software Production & Test	1000K
20.	TSOS + VMM	2800K
21.	TSOS + MPOS + VMM	1400K
	TOTAL	25,970K

TABLE 18 - SCHEDULE FOR CORRECTION OF IBM 360/370 SOFTWARE DEFICIENCIES (based upon $2 million per year)

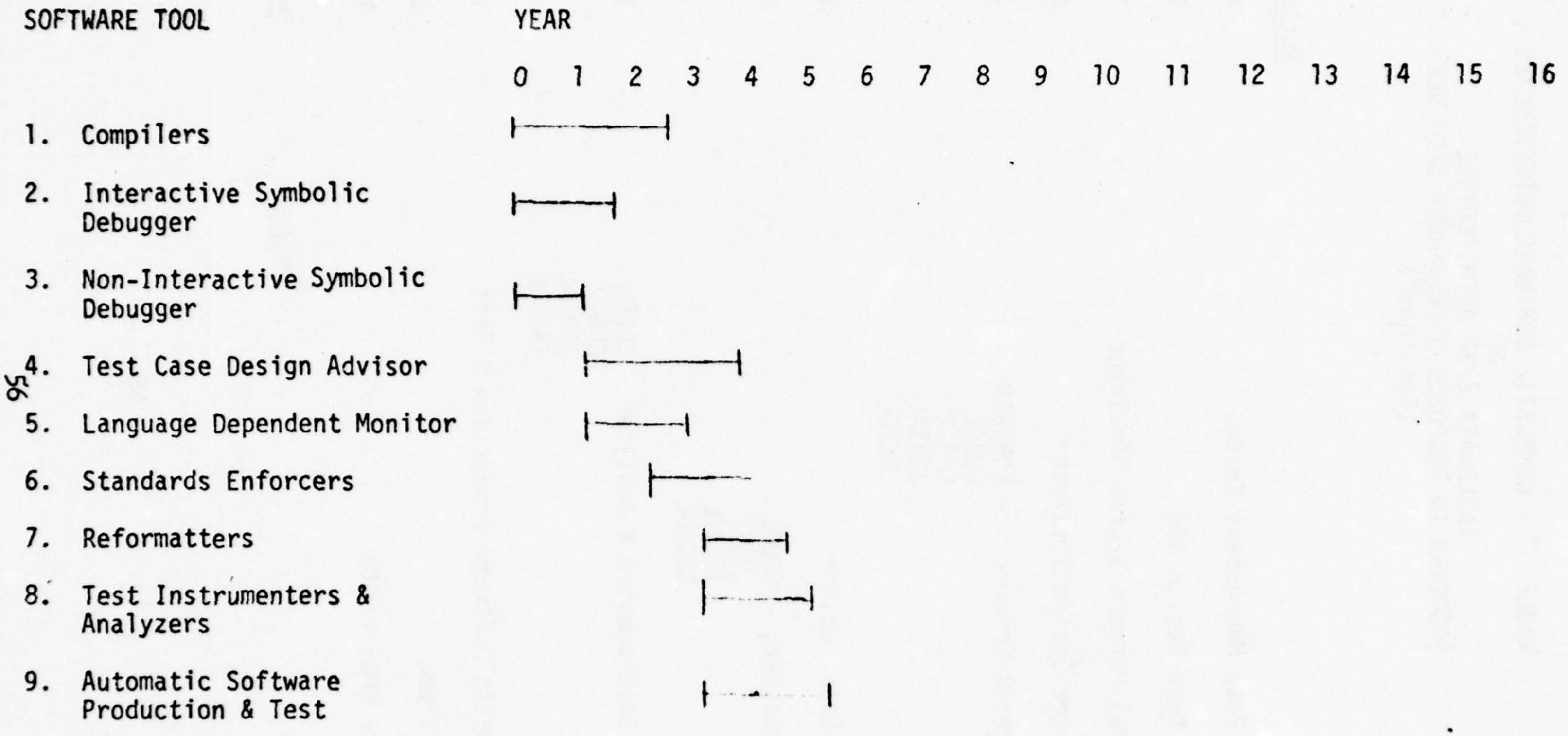


TABLE 19 - SCHEDULE FOR CORRECTION OF DEC PDP-11 SOFTWARE DEFICIENCIES (based upon $2 million per year)

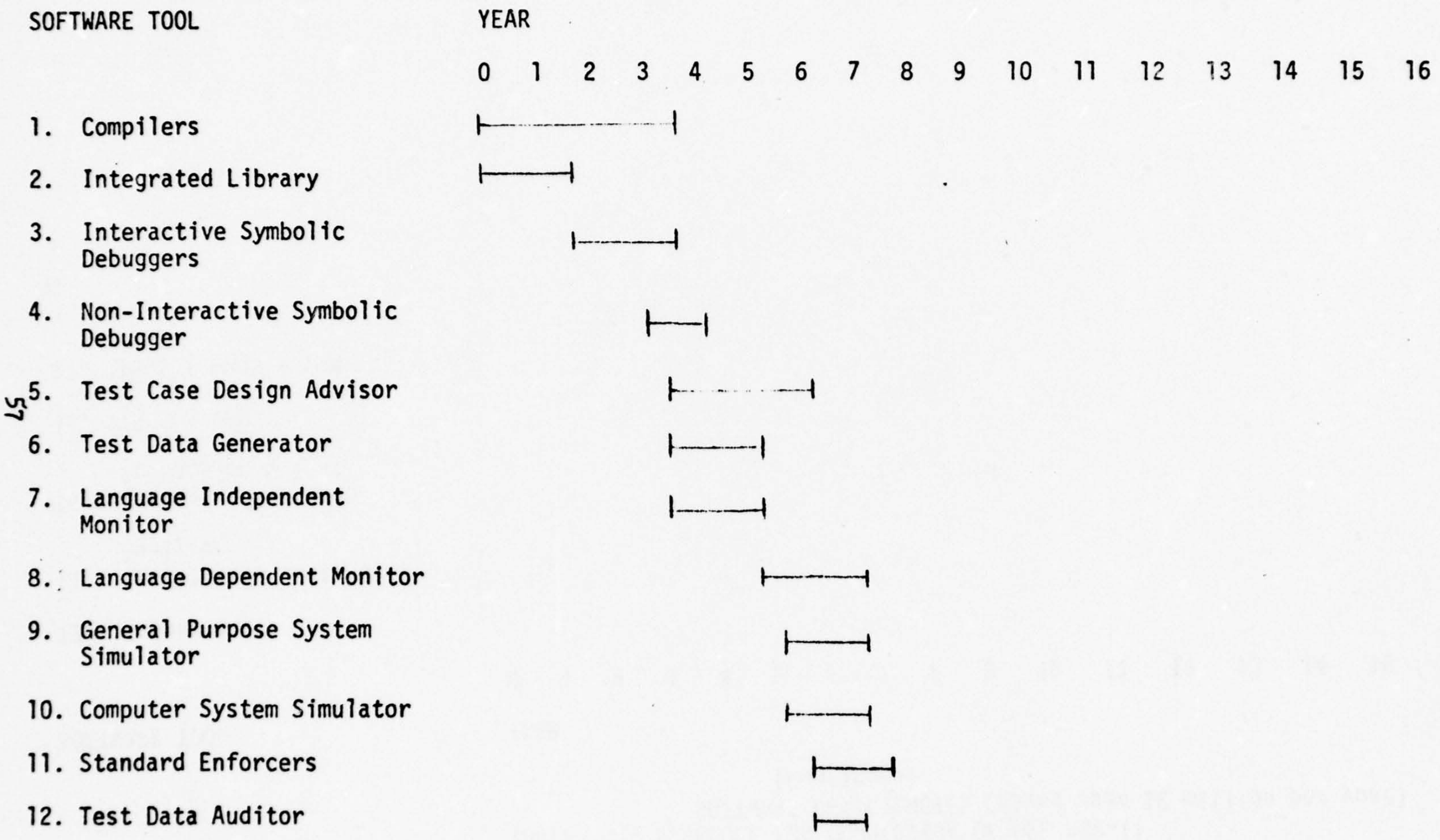


TABLE 19 - SCHEDULE FOR CORRECTION OF DEC PDP-11 SOFTWARE DEFICIENCIES (based upon $2 million per year) (Continued)

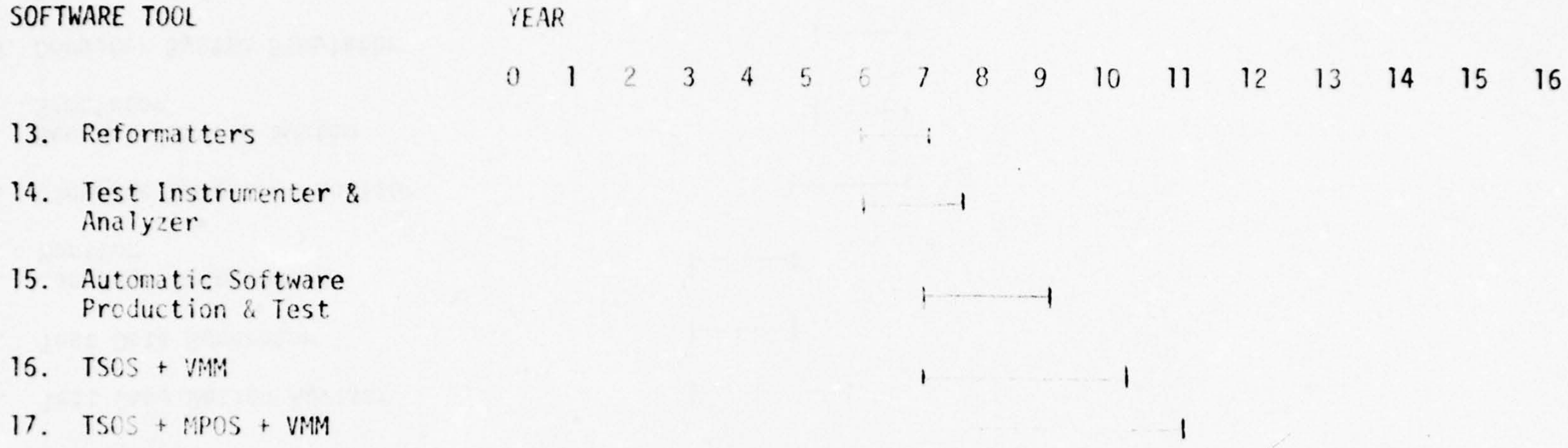


TABLE 20 - SCHEDULE FOR CORRECTION OF INTERDATA 8/32 SOFTWARE DEFICIENCIES (based upon $2 million per year)

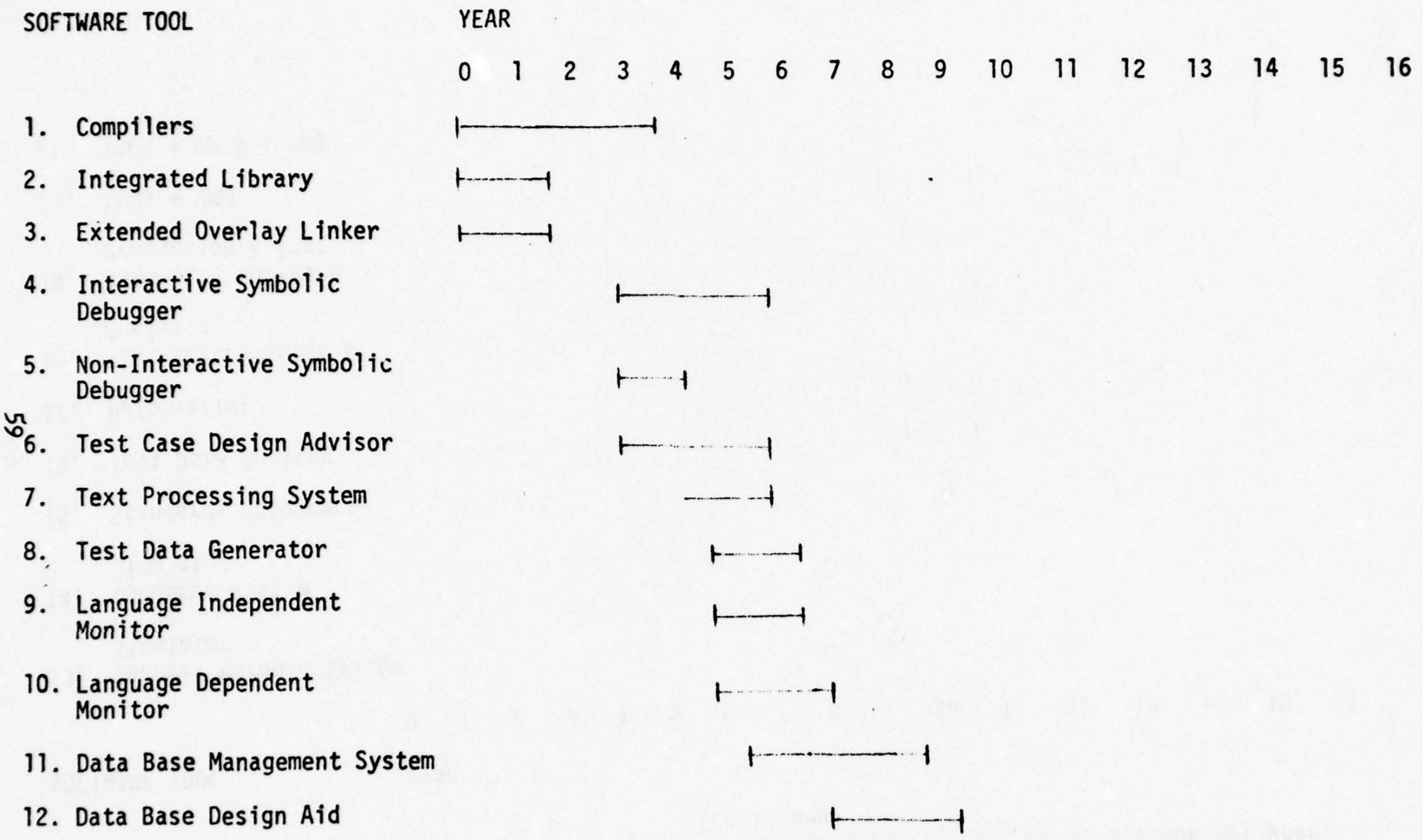


TABLE 20 - SCHEDULE FOR CORRECTION OF INTERDATA 8/32
SOFTWARE DEFICIENCIES (based upon $2 million per year)
(Continued)

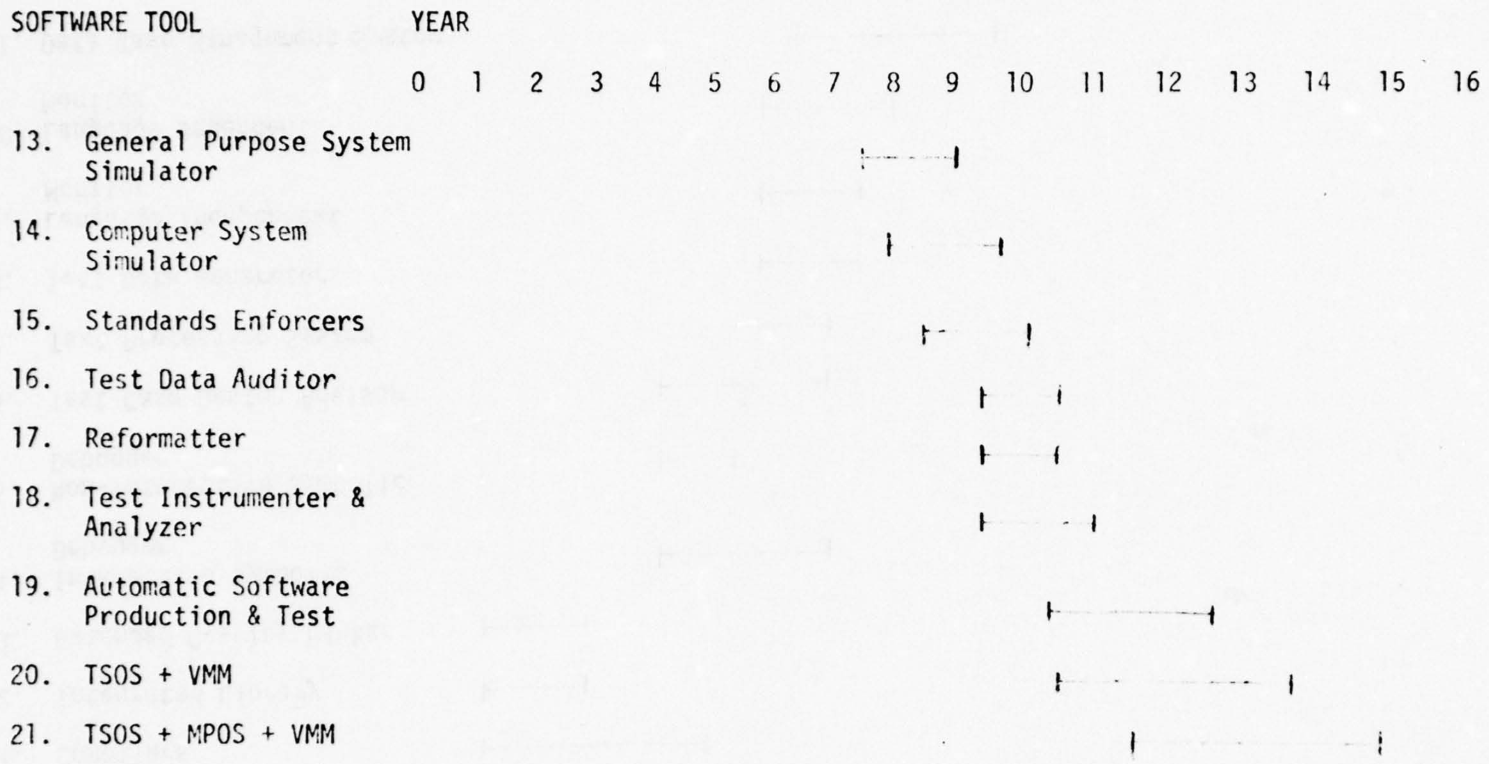


TABLE 21 - SCHEDULE FOR CORRECTION OF IBM 360/370 SOFTWARE DEFICIENCIES
(BASED UPON $3 million per year)

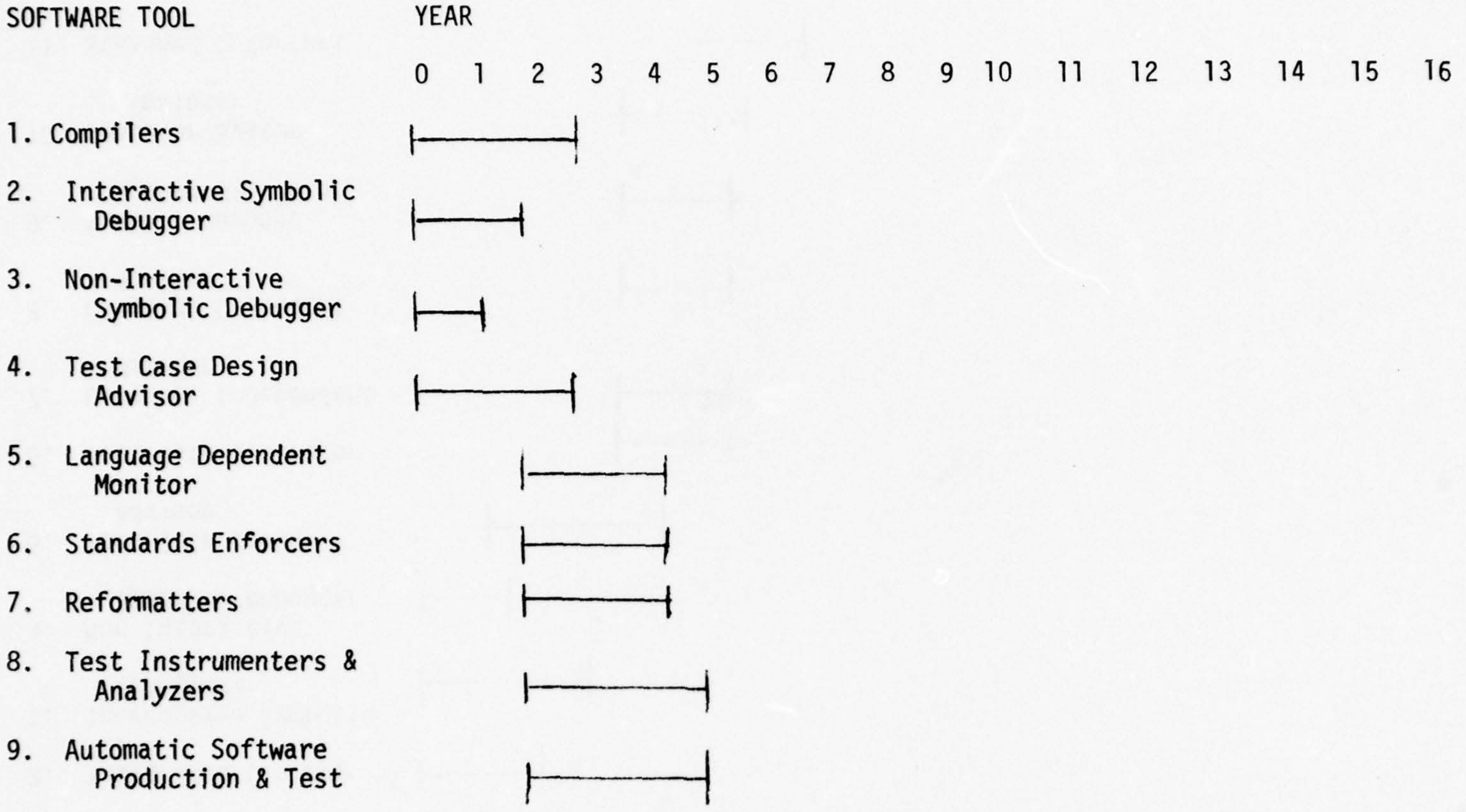


TABLE 22 - SCHEDULE FOR CORRECTION OF DEC PDP-11 SOFTWARE DEFICIENCIES
(Based upon $3 million per year)

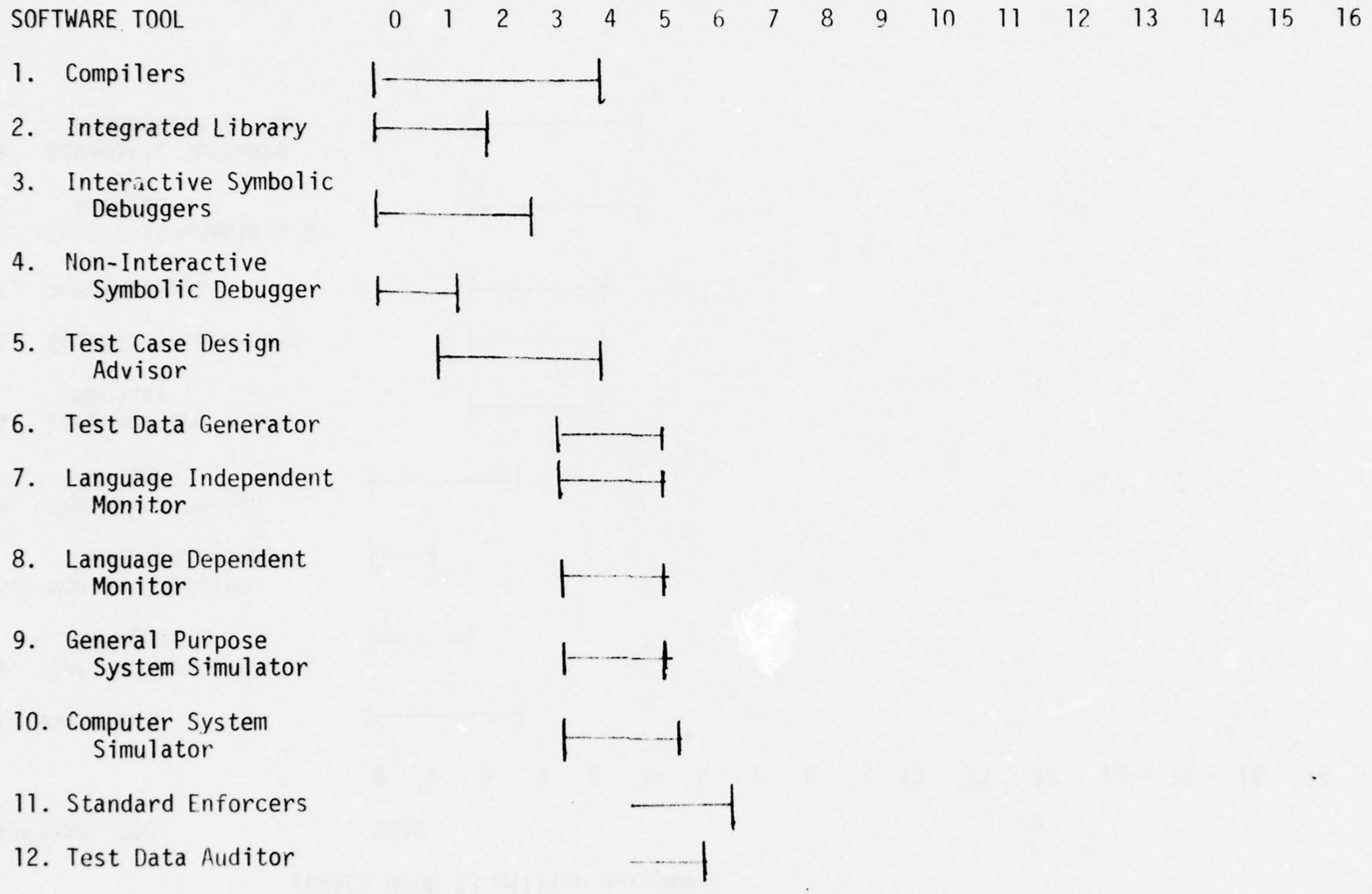


TABLE 22 - SCHEDULE FOR CORRECTION OF DEC PDP-11 SOFTWARE DEFICIENCIES
(Based upon $3 million per year) (Continued)

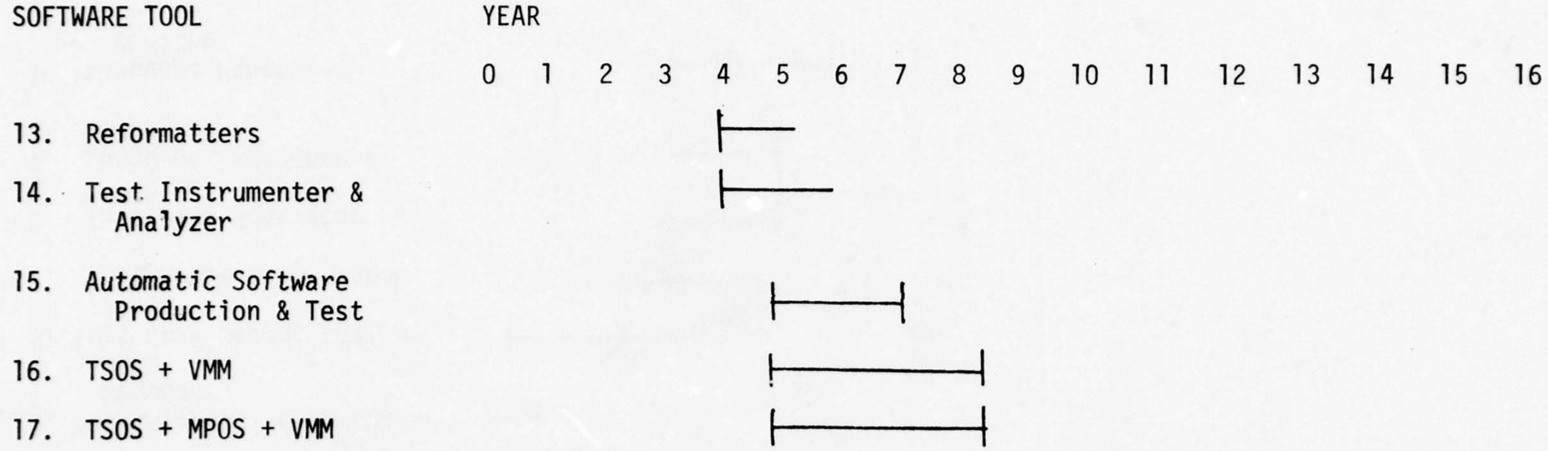


TABLE 23 - SCHEDULE FOR CORRECTION OF INTERDATA 8/32 SOFTWARE DEFICIENCIES
(based upon $3 million per year)

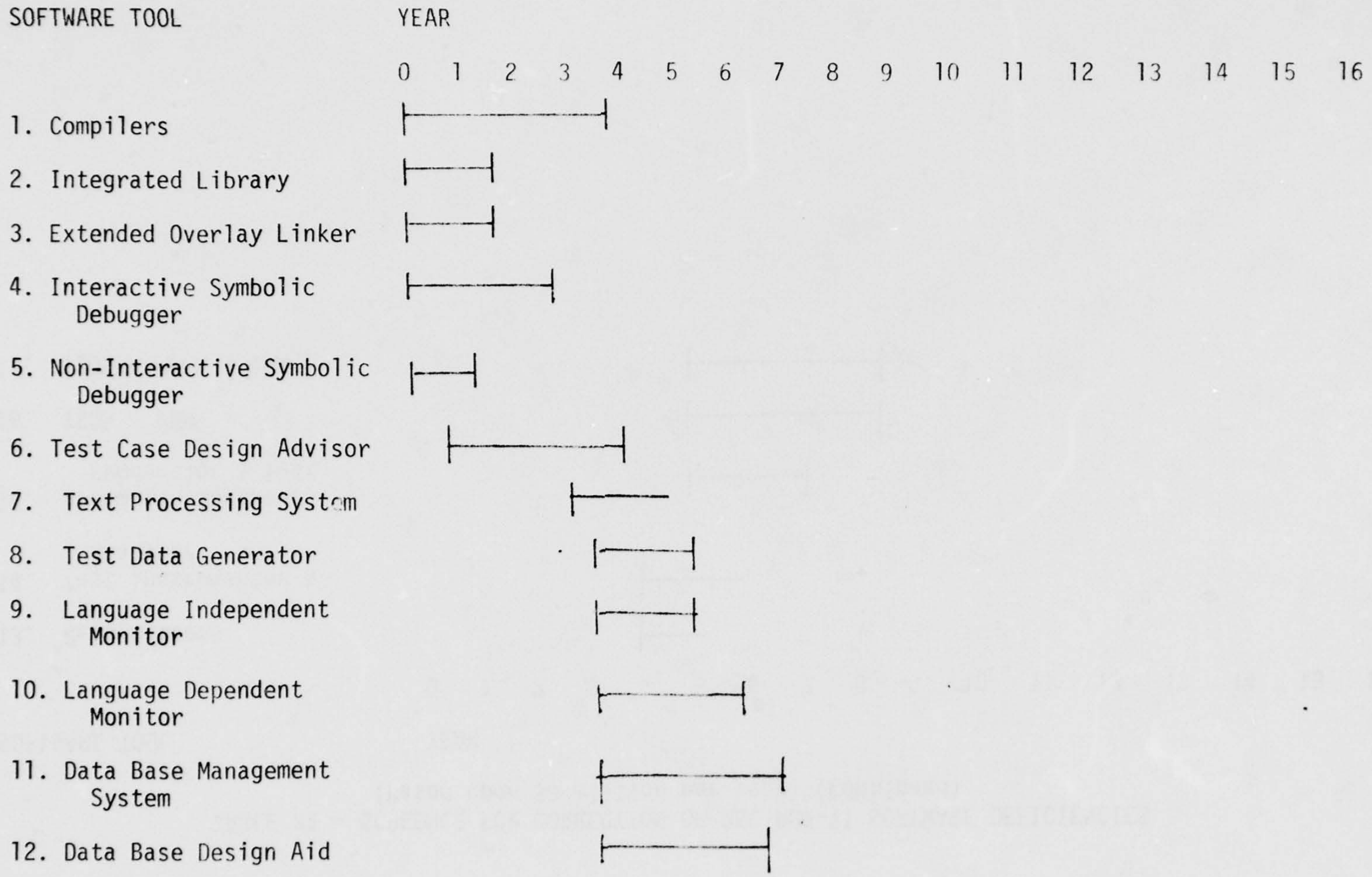


TABLE 23 - SCHEDULE FOR CORRECTION OF INTERDATA 8/32 SOFTWARE DEFICIENCIES
(based upon $3 million per year) (Continued)

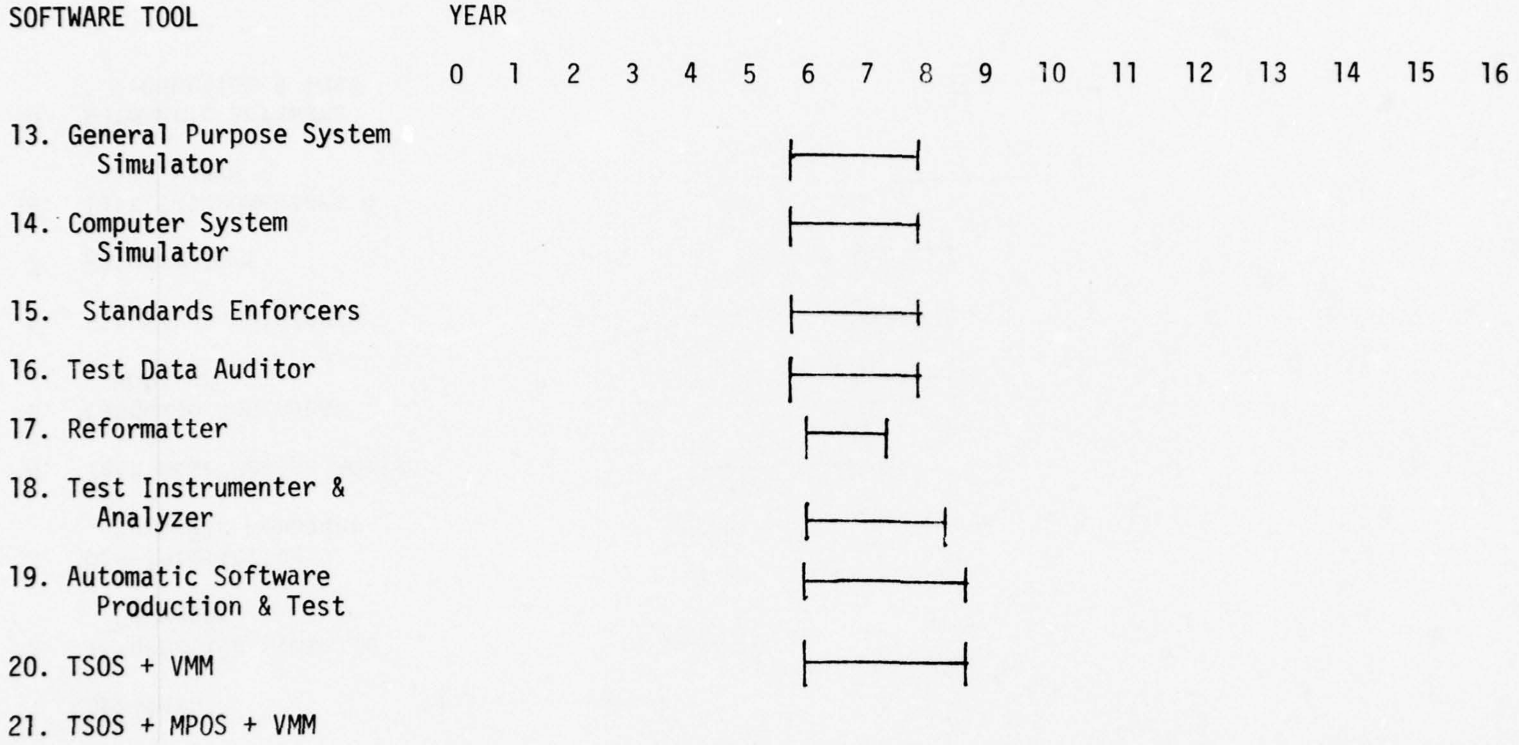


TABLE 24 - SCHEDULE FOR CORRECTION OF IBM 360/370 SOFTWARE DEFICIENCIES
(based upon $1 million per year)

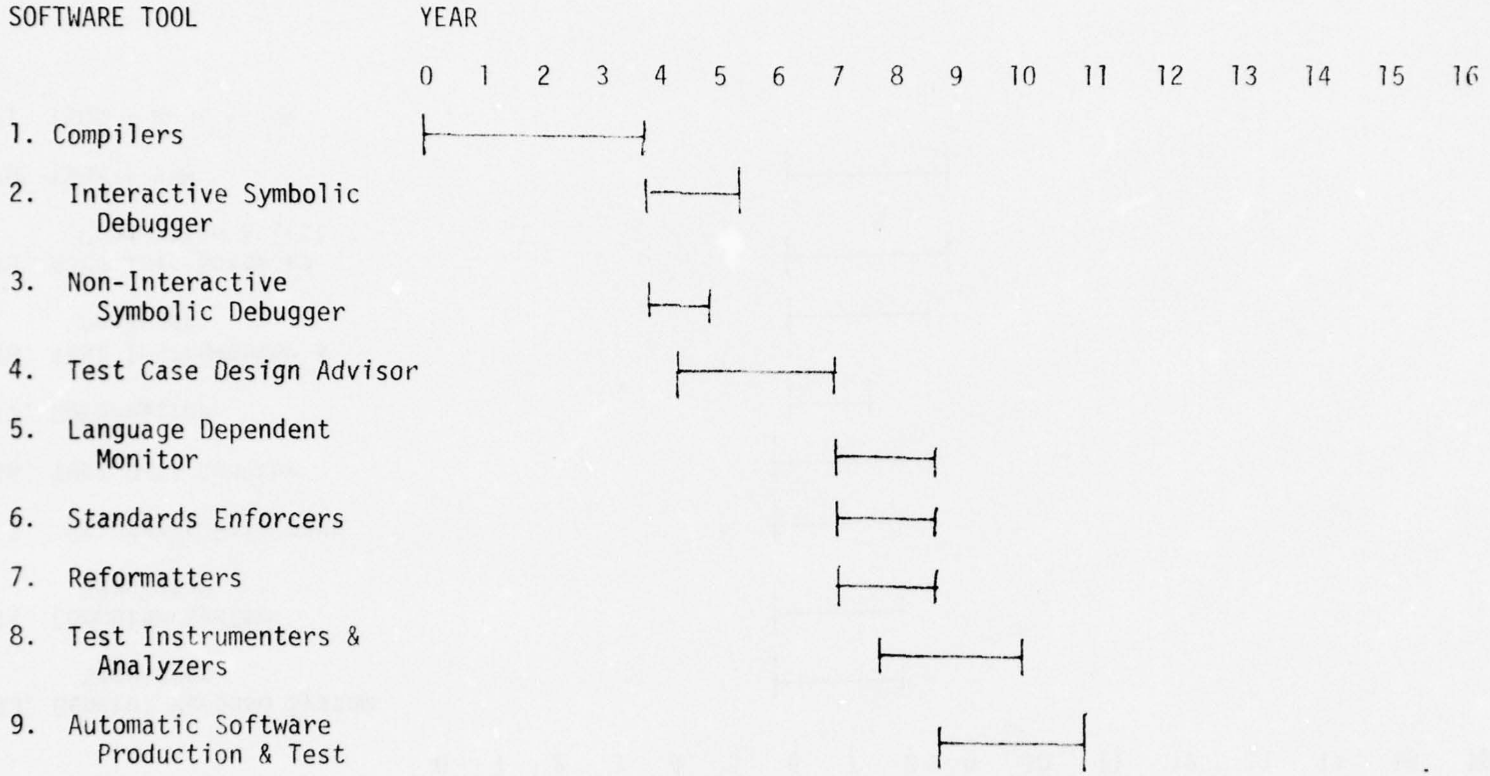


TABLE 25 - SCHEDULE FOR CORRECTION OF DEC PDP-11 SOFTWARE DEFICIENCIES
(based upon $1 million per year)

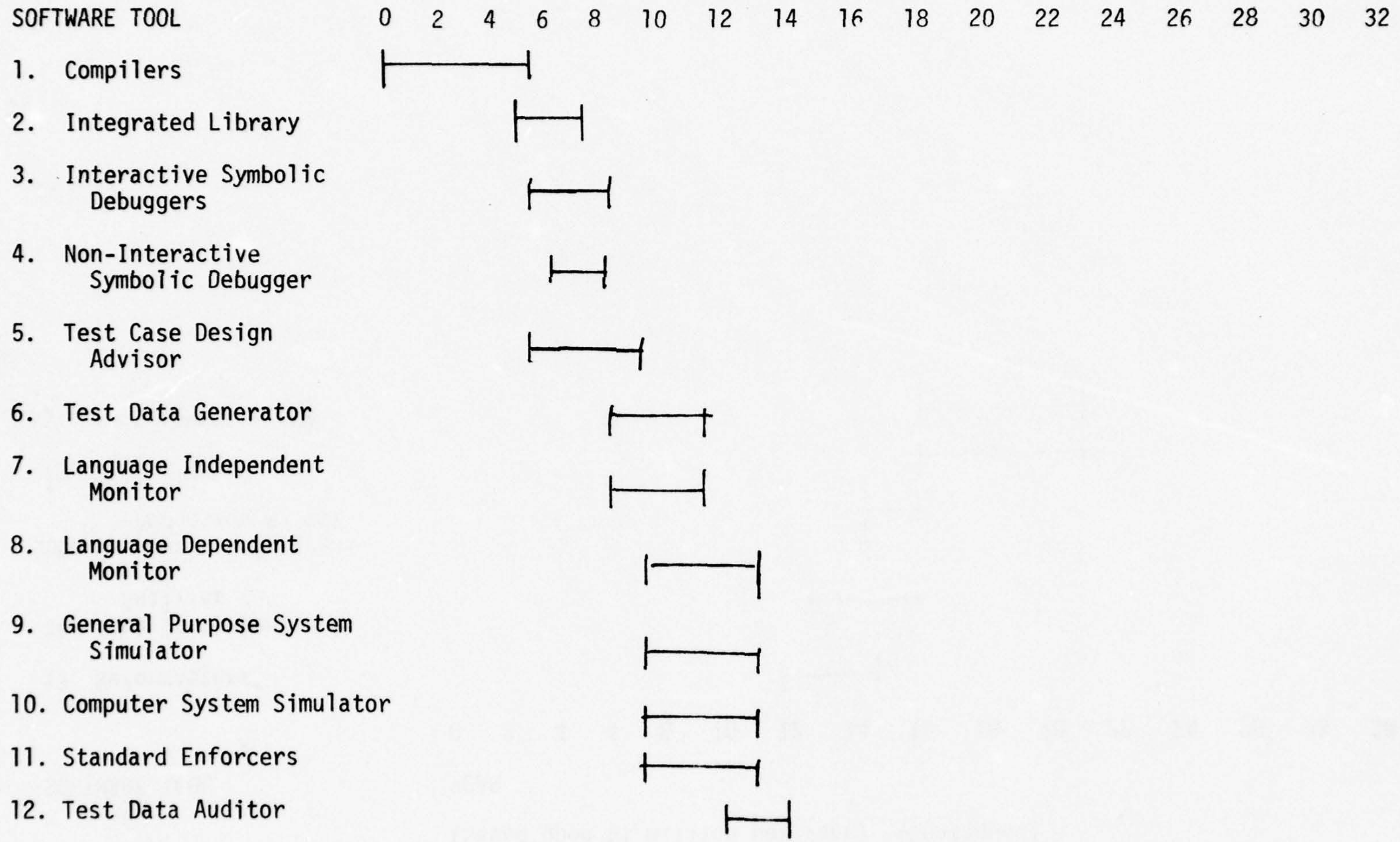


TABLE 25 - SCHEDULE FOR CORRECTION OF DEC PDP-11 SOFTWARE DEFICIENCIES
(based upon $1 million per year) (Continued)

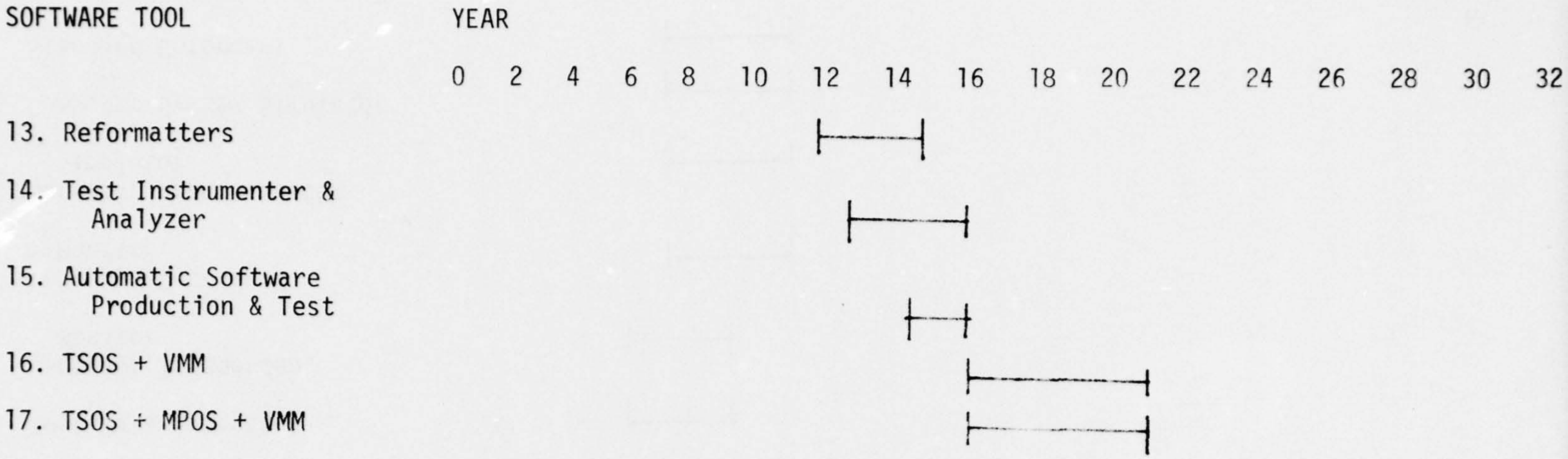


TABLE 26 - SCHEDULE FOR CORRECTION OF INTERDATA 8/32 SOFTWARE DEFICIENCIES
(based upon $1 million per year)

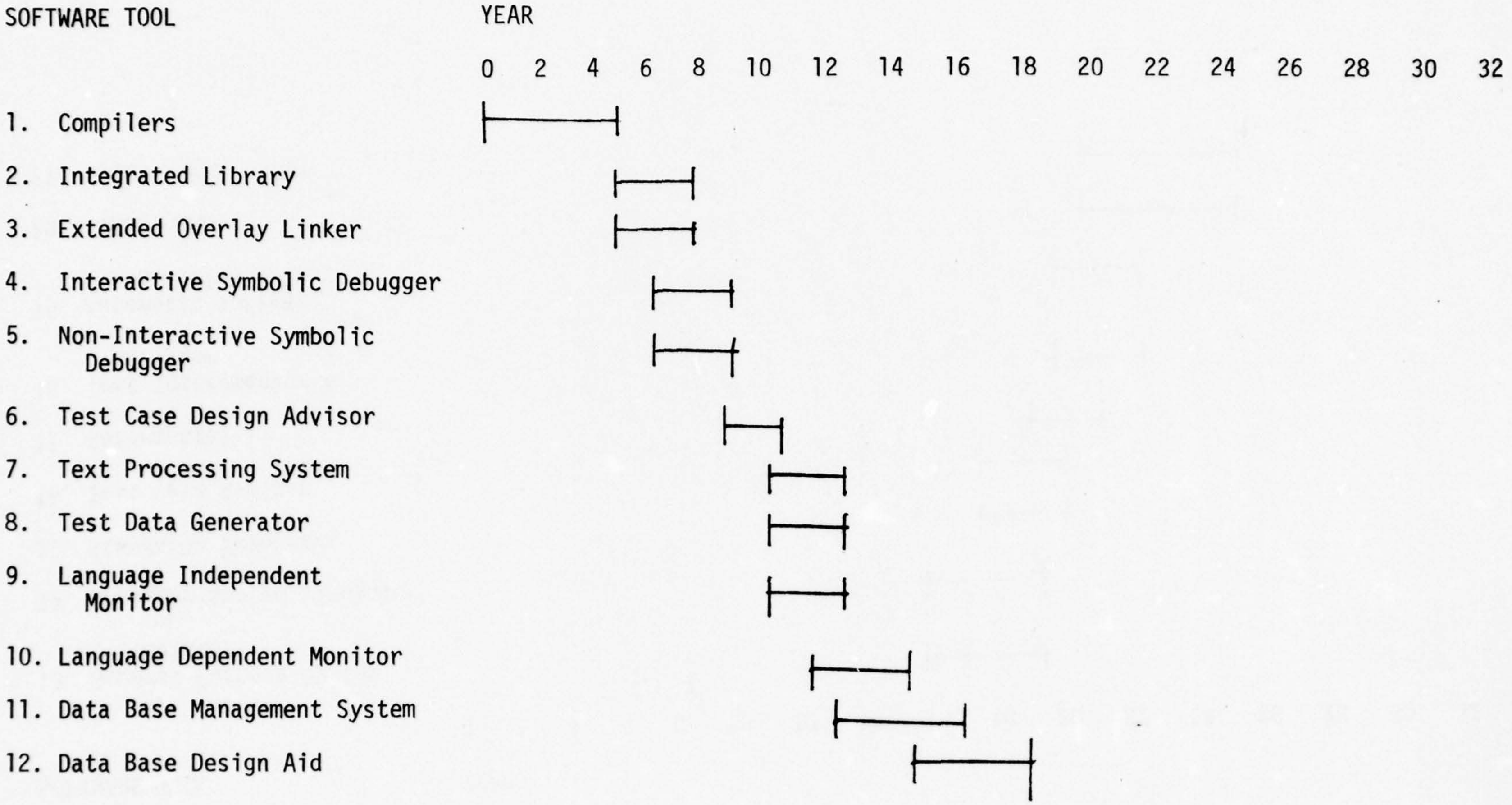
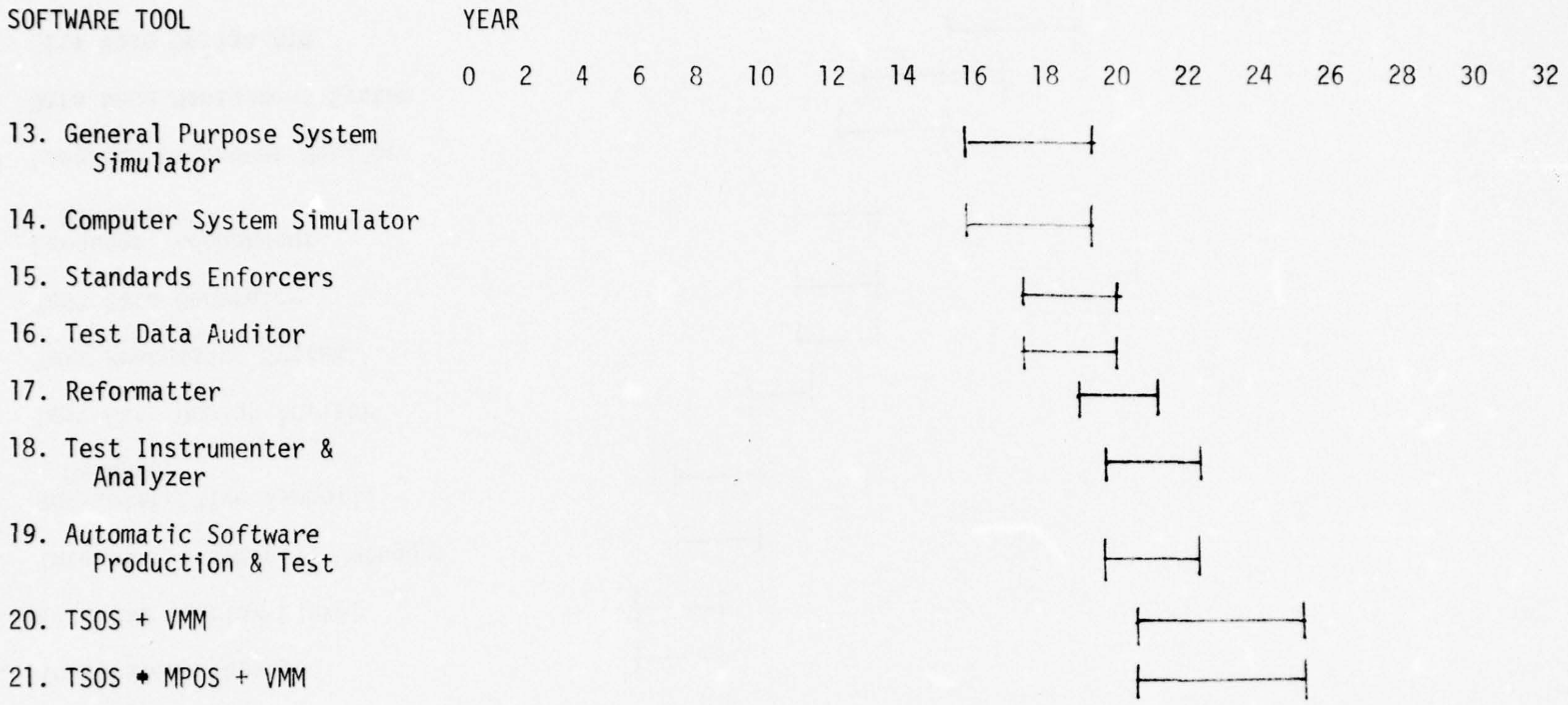


TABLE 26 - SCHEDULE FOR CORRECTION OF INTERDATA 8/32 SOFTWARE DEFICIENCIES
(based upon $1 million per year) (Continued)



APPENDIX A - DESCRIPTION OF SOFTWARE BASE COMPONENTS

A.1 Introduction

This appendix contains descriptions of each of the software tool categories recommended for inclusion in the evaluation of the software base. One purpose of these descriptions was to provide the Selection Committee with a sufficient definition to decide on the merits of the tool when allocating applicability points and evaluating the availability of tools. The descriptions attempt to establish a minimum level of features or capabilities required of a tool in order for it to qualify for acceptance. The intent is to establish a threshold to differentiate usable tools from those that are not.

The tool descriptions are organized as illustrated in Tables A-1 to A-3 along the structure presented in Section 3.3. First, the layer 3 tools are presented grouped under primary phase of the development process which the tools support. Next, the layer 2 tools are presented grouped under the functions that they support. Layer 2 functions are generally common to all the development activities/phases. Finally, selected features of layer 1 tools (operating systems) are discussed. The operating systems features selected represent a judgment of high cost capabilities which significantly affect the effectiveness of the development environment.

TABLE A-1

Layer 3 Tool List

- 1.1 Requirements Analysis
  - 1.1.1 General Purpose System Simulators
  - 1.1.2 System Description Languages and Analyzers
- 1.2 Software Design
  - 1.2.1 Computer System Simulators
  - 1.2.2 Data Base Design Aids
  - 1.2.3 Data Dictionary Systems
- 1.3 Construct System Tests
  - 1.3.1 Test Data Generators
  - 1.3.2 Test Data Auditors
  - 1.3.3 Test Case Design Advisors
  - 1.3.4 Test Instrumenters and Analyzers
- 1.4 Build and Unit Test
  - 1.4.1 Assemblers
    - 1.4.1.1 Basic Assembler
    - 1.4.1.2 Macro Assembler
  - 1.4.2 Compilers
  - 1.4.3 Linkers
    - 1.4.3.1 Basic Linker
    - 1.4.3.2 Simple Overlay Linker
    - 1.4.3.3 Extended Overlay Linker
  - 1.4.4 Debugging Aids
    - 1.4.4.1 Interactive Symbolic Debugger
    - 1.4.4.2 Interactive Absolute Debugger
    - 1.4.4.3 Non-Interactive Symbolic Debugger
    - 1.4.4.4 Non-Interactive Absolute Debugger
  - 1.4.5 Module Libraries & Change Control Systems
    - 1.4.5.1 Basic or Loosely Coupled Libraries
    - 1.4.5.2 Integrated Libraries
    - 1.4.5.3 Automated Software Production and Test

1.4.6 Performance Monitors

- 1.4.6.1 Language Dependent Monitors
- 1.4.6.2 Language Independent Monitors

1.4.7 Standards Enforcers

1.4.8 Preprocessors/Reformatters

- 1.4.8.1 Preprocessors
- 1.4.8.2 Reformatters

TABLE A-2

Layer 2 Tool List

2.1 Data Base Management Systems

2.2 Project Management Aids

2.2.1 PERT/CPM Systems
2.2.2 Project Estimation Systems

2.3 Documentation Aids

2.3.1 Text Processing Systems
2.3.2 Flowchart Construction Languages
2.3.3 Automatic Flowcharters

2.4 Data Manipulation Utilities

2.4.1 Sort/Merge
2.4.2 Editors

2.4.2.1 Interactive Source Language Editors
2.4.2.2 Interactive Object Module Editors
2.4.2.3 Batch Source Language Editors
2.4.2.4 Batch Object Module Editors

2.5 Information Retrieval Systems

2.5.1 Query Languages
2.5.2 Report Writers

TABLE A-3

Layer 1 List/Features

3.1 Basic Operating Systems (BOS)

3.2 Multiprogramming Operating Systems (MOS)

3.3 Multiprocessor Operating Systems (MPOS)

3.4 Virtual Machine Monitors (VMM)

3.5 Time Sharing Operating Systems (TSOS)

3.6 Real-Time Operating Systems (RTOS)

A.1.1 Layer 3 Tools

A.1.1.1 Tools for Requirements Analysis

A.1.1.1.1 General Purpose System Simulator

This tool allows a user to construct a computer model of a real or proposed system and to perform simulation experiments to determine the behavior of the model under various operating conditions. A "general purpose" simulator provides the modeler with very basic building blocks or functions that can be parameterized to represent computers, magnetic storage, teleprocessing lines, manual operations, etc. These functions of building blocks usually have names like "queue," "server," "storage," "clock," and so on, and the simulation is controlled by the occurrence of discrete events. A general purpose simulator must provide the following services:

(1) Allow the user to define the static building blocks that represent the active (processing) and passive (queue and storage) components of the system. The user must be able to define both the interconnection of system elements and their capacities or processing rates.

(2) Allow the user to specify the rate at which transactions are generated to drive the system. Common arrival distributions including random, poisson, exponential and hyperexponential should be available.

(3) Allow the results of a simulation to be clearly and easily displayed both during the simulation and at its end. The user must be able to selectively display critical transaction arrival rates, storage utilizations, processor utilizations, and queue length and delay statistics.

A.1.1.1.2 System Description Languages and Analyzers

Assist system analysts in describing the functional characteristics of a system, and in validating the consistency and completeness of a functional decomposition. The System Description Language should allow for application of a top-down methodology. It should provide language facilities to define both processes or activities and data. It should provide facilities to quantify properties of both processes and data. Such quantifiable properties may be used to describe performance, size, weight, cost, etc. It should also provide facilities to define the dynamic behavior of the system in sufficient detail to enable the generation of input to simulation systems.

A large part of the value of System Description Languages and Analyzers is determined by the types of reports it produces to assist the analysts in performing their function, communicating with other analysts, and communicating with users and management. Representative capabilities that should be present in available reports are:

(1) The ability to generate a structured description of any process (i.e., constituent subprocesses, sub-subprocesses, etc.)

(2) The ability to generate a structured description of any datum.

(3) The ability to obtain all attributes of a process or datum.

(4) The ability to obtain all uses of a datum.

(5) The ability to obtain all activation conditions of a process.

(6) The ability to obtain all processes that may be activated by an event.

(7) The ability to check that all data used is created somewhere within the system or is a system input.

(8) The ability to check that all data generated within the system is used somewhere or is a system output.

A.1.1.2 Tools for Software Design

A.1.1.2.1 Computer System Simulator

This tool is similar in nature to the general purpose simulator, except that its basic building blocks represent real computer system components whose modeled behavior approximates the throughputs, capacities, and access times achievable on the modeled equipment. The model should contain both hardware characteristics for processor speeds, storage, size, access time, etc., but should have available analytical models of the behavior of commonly used support software. In order to be useful to the software designer, the computer system simulator should possess the following capabilities: (1) It should be easy to construct a model of any legally configurable group of hardware components. (2) It should be possible to validate the results of a simulation on a subset of the available hardware. (3) It should be possible to alter the modeled behavior of particular components to correspond to different software algorithms. (4) "Off-the-shelf" models for common storage devices, access methods, operating systems, and so on should be available.

A.1.1.2.2 Data Base Design Aids

These tools assist the data base designers in grouping data elements into logical record classes and in determining the relationships among logical record classes implicit in either the nature of the data or the usage of the data. The tool should also provide assistance in evaluating the performance tradeoffs among possible physical structures for a given logical data base structure using, as a minimum, frequency of usage, size, and number of occurrences data, and a variety of access methods and storage placement strategies. The tool should be capable of handling one-to-one, one-to-many, and many-to-many relationships among logical record classes.

A.1.1.2.3 Data Dictionary Systems

These tools assist data base designers in managing the data definition activities. During succeeding phases of development, a data dictionary system provides services to the data base manager to assist in controlling the usage activities. Data dictionaries define data elements by associating with the data element name a set of properties such as data type, external representation, units, value set, textual data, etc. They assist in controlling the usage by identifying logical or physical record classes containing the data element and by maintaining the history of changes to the data element definition.

The tool typically provides a selective retrieval capability. Additional features include processors to generate data declarations or high level languages and processors to generate data editing and validation programs. Some COMPOOL facilities can be classified as basic data dictionary systems.

A.1.1.3 Tools for the Construction of System Tests

A.1.1.3.1 Test Data Generators

Test data generators create data files for testing and validating computer programs. The test data file may be used directly as input to the module under test or may be used as input to a driver program which converts the test data file to suitable stimuli for the module under test. Available commercial test data generators may be directly applicable to military tactical, and command and control applications. A test data generator should have the following capabilities:

(1) Create files of all storage organizations supported by the host operating system, i.e., sequential, index sequential, random, etc.

(2) Create files of all intra-file structures supported by the host operating system, i.e., fixed and variable length records, hierarchical record organizations, etc.

(3) Accept a suitable description of data elements and their value set.

(4) Accept a suitable description of the intrarecord structure.

(5) Be capable of generating records with valid and invalid (i.e., within or outside the value set) data elements under the control of the test specifier.

(6) Provide control over the numbers and type of records generated.

(7) Provide control over the logical sequencing of records within the file.

(8) Provide control over the physical placement of records within the file.

A.1.1.3.2 Test Data Auditors

These tools compare data files against specifications and produce reports of discrepancies and/or compliance. The specifications may be in the form of another data file or in the form of file and record descriptions compatible with a companion Test Data Generator tool. This tool category is not intended to include simple file comparison programs of the type which perform a character by character or word by word comparison. To be included in this category a Test Data Auditor should have the following capabilities.

(1) Accept input files of all storage organizations supported by the host operating system, i.e., sequential, index sequential, random, etc.

(2) Accept input files of all intrafile structures supported by the host operating system, i.e., fixed and variable length records, hierarchical record organizations, etc.

(3) Accept a suitable description of data elements and optionally their value set.

(4) Accept a suitable description of the intrarecord structure.

(5) Provide control over the number and type of records audited.

(6) Provide control over the data elements to be audited and optionally over the type of comparison to be performed.

A.1.1.3.3 Test Case Design Advisors

These tools analyze programs written in a high level language and present the results of that analysis in a form suitable to assist test case designers in the selection of test data. Suitability of presentation of analysis results depends on the test data selection criterion. There is no known test data selection criterion which is guaranteed to yield a complete set of tests, i.e., a set of tests which, if successfully passed, assures the correctness of a program. However, there are known types of errors which will not be detected unless the test data selection criterion satisfies certain conditions. The simplest such condition is that the test set exercises all statements in a program. Otherwise, errors present in the unexercised program segments could not be found. A somewhat more comprehensive criterion is to exercise all branch conditions in a program. Such criterion will not only exercise all statements but also additional combinations of statement sequences. Another useful test data selection criterion is boundary or limit test data. Such test cases assist in verifying that the program performs correctly within the valid range of inputs and that it adequately protects itself from invalid data. This criterion is also useful in verifying that the special cases typically present at the boundaries are adequately handled.

Test case design analyzer should support one or more of the above test data selection criteria. As a minimum it should:

(1) Identify all linear program segments, i.e., all sequences of statements such that if Si and Sj are any two statements in the sequence, execution of Si implies execution of Sj and vice versa.

(2) Identify predicates that must be satisfied to exercise every linear segment.

(3) Provide assistance in tracing the predicate variables to the module's external interfaces.

(4) Provide assistance in identifying valid combinations of predicates, i.e., under what circumstances a predicate is a false tautology.

A.1.1.3.4 Test Instrumenters and Analyzers

These tools instrument modules under test so as to collect data characterizing the behavior of the module. The collected data is post-processed by an analyzer which produces reports to aid in determining the success of the test in satisfying a test data selection criterion. The tool should support one or more of the test data selection criteria discussed under Test Case Design Advisors. As a minimum it should identify all linear segments of the module under test not covered by the test.

A.1.1.4 Tools for Building & Unit-Testing Software

A.1.1.4.1 Assemblers

Assemblers allow programs to be coded in a symbolic language in which statements generally correspond to a single machine instruction. Allow machine instructions and storage areas for variable and constant information to be given symbolic labels, and permit the use of these labels in assembler statements. Permit the insertion of commentary information on the same statement as a coded instruction or data definition, or on a separate statement. In addition to these functions, assemblers may have other capabilities as outlined below.

A.1.1.4.1.1 Basic Assembler

To be minimally useful, the basic assembler must:

(1) Have a minimum label size of 6 characters.

(2) Allow a label to be used in a statement before it has been defined.

(3) Allow the explicit declaration of external symbols and entry points.

(4) Have a symbolic equivalent for the value of the current location counter.

(5) Produce a symbol cross reference listing.

(6) Produce error diagnostics that identify the statement and field in error.

(7) Produce a listing with both symbolic and machine language instructions.

A.1.1.3.1.2 Macro Assembler

This tool provides the features of a basic assembler, plus the ability to define named groups of multiple basic assembly language statements, or MACROS, that can be inserted in line in an assembly language program simply by writing the name of the macro. In addition to this simple code substitution capability, a macro assembler should have the following capabilities:

(1) The ability to modify the symbols in the macro instructions by supplying keyword or positional parameters with the macro when it is invoked.

(2) The ability to modify individual fields within a single instruction in the macro.

(3) The ability to conditionally generate or bypass code sequences within the macro based upon values passed through its parameter list.

(4) The ability to allow the macro processor to check the validity of the parameter list coded by the invoker of the macro, and to issue warnings when erroneous invocations occur.

(5) The ability to invoke an inner macro within the definition of an outer macro.

(6) The ability to control whether or not the expansion of a macro is printed.

A.1.1.4.2 Compilers

Compilers translate programs written in a high level language into either relocatable object code acceptable to a linker or assembly language acceptable to an assembler. The source language accepted by the compiler must be a subset or dialect approved by the cognizant DoD agency. Where validation standards have been established (e.g., COBOL, JOVIAL), the compiler must have successfully passed the validation test as interpreted by the validation agency. The requirement of relocatable object code or assembly language output is not intended to exclude compilers which do not produce directly executable code, i.e., interpretive code. Cross compilers for an architecture other than the CFA under evaluation are excluded. In addition to the primary object code output, the compiler should provide the following capabilities preferably under option control:

(1) Source listing

(2) Symbol dictionary listing including symbol attributes

(3) Cross reference listing

(4) Suitable error diagnostics

(5) Debugging aids support

A.1.1.4.3 Linkers

Linkers combine the text produced by separate invocations of compilers and assemblers ("object modules") into executable code strings ("load modules" or "core images") that can be loaded into the computers main storage and executed without further pre-processing.

A.1.1.4.3.1 Basic Linker

The basic linker must perform at least the following functions:

(1) Allow the user to supply control statements that can control:

(a) the absolute main storage address that is to be the origin of the resulting load module

(b) the sequence in which the input object decks are to be loaded

(c) the symbolic name of the resulting load module

(2) Resolve all external symbol references.

(3) Relocate all required addresses that were specified relative to the load point.

(4) Produce error diagnostics that list any unresolved external symbols.

(5) Produce a load module map that shows the absolute address of all external symbols, entry points, and the boundaries of the consistent object modules.

(6) Produce a cross reference listing for all external references and entry points, including external symbols that are unreferenced.

A.1.1.3.3.1 Simple Overlay Linker

In addition to the services of the basic linker, the simple overlay linker allows the user to define phases or overlays of load module code in which certain object modules are linked into the same physical main storage locations. In the case of the simple overlay linker, only the resolution of external symbol references and the relocation of address references is handled; controlling the contents of storage at execution time is the responsibility of the user program. The simple overlay linker provides the following additional services:

(1) Ability to define a "root phase," or a group of object modules that are linked into a load module that is always resident in main storage and is never overlayed by other code.

(2) Ability to specify which object modules are to be grouped into "overlay phases" that may be read into the same area of storage at the direction of the root phase usually via calls to supervisor functions.

(3) Ability to specify a simple tree of segments of overlays.

A.1.1.4.3.2 Extended Overlay Linker

The major extensions this tool has over the simple overlay linker is the addition of the following capabilities:

(1) The ability to assist the user with storage content management by automatically detecting and routing inter-segment calls or transfers of control through the supervisor in order to invoke the loading of the required phases.

(2) The ability to define more complex overlay trees that may consist of several discontiguous regions.

(3) The ability to "edit" object and load modules by rearranging, replacing or deleting their constituent object modules.

(4) The ability to perform automatic module library searches in order to resolve external references.

(5) The ability to record extensive information about the characteristics of the load module (e.g., whether or not it is reentrant, whether it is executable, etc.).

A.1.1.4.4 Debugging Aids

These tools assist the programmer in locating the sources of program errors that have been discovered during unit testing, usually by giving him some control over the execution of the module under test that is external to the normal program code. Debugging aids can be classed as symbolic or absolute and as interactive and non-interactive. Symbolic aids allow the programmer to manipulate the instructions and data in the load module under test by using the symbolic labels and data names in the source program (assembler or HOL) that were used to create the load module. Symbolic debugging aids are always language dependent and they require that the compiler or assembler produce a machine readable symbol table. Absolute debugging aids require that the programmer reference his module by using absolute addresses; however, they may be used in cases where the source language translator does not supply a symbol table. Interactive debugging aids are designed for use in a time sharing environment. They allow the programmer to debug his program as if he were controlling the computer through the switches on a control panel. Non-interactive debugging aids require that debugging control be enforced via control cards in the batch job stream. Minimum requirements for each type of debugging aid are contained in the following section.

A.1.1.4.4.1 Interactive Symbolic Debuggers

(1) Must give control to the user prior to initiation of the module under test,

(2) Must allow all labeled data and instructions to be displayed and altered,

(3) Must allow the insertion of breakpoints,

(4) Must have a single step execution mode,

(5) Must allow the program to be restarted after a breakpoint either at the next instruction or at a different location,

(6) Must allow the user to obtain on-line or off-line snapshot or full storage dumps,

(7) Must allow the display and alteration of machine facilities (as opposed to symbolic locations and variables) such as status words, index registers, etc.

One Iterative Symbolic Debugger should be evaluated for each source language.

A.1.1.4.4.2 Interactive Absolute Debugger

Requirements for this tool are the same as for the symbolic debugger, except that references to program and data storage are by absolute or relative main storage address.

A.1.1.4.4.3 Non-Interactive Symbolic Debugger

Because the program under test will not be under the direct control of a programmer when this tool is in use, it must have more extensive options than the interactive version. In addition to all the display, alteration and restart capabilities mentioned above, the non-interactive symbolic debugger must:

(1) Allow the setup of calling parameter lists, global variables, and so on prior to module execution,

(2) Allow the selective invocation of instruction traces for all or parts of a program. In order to restrict the volume of output, the amount of data displayed should be optional, and the invocation/termination of the trace should be controllable based on loop counts or the value of variables in the program under test.

A.1.1.4.4.4 Non-Interactive Absolute Debugger

See paragraphs A.1.1.4.4.2 and A.1.1.4.4.3 above.

A.1.1.4.5 Module Libraries & Change Control Systems

These tools provide computer controlled maintenance of groups of related source modules (programs), object modules (the output of assemblers and compilers), and load modules (the output of linkers). Depending upon their sophistication, they may also enforce various consistency checks, as described below.

A.1.1.4.5.1 Basic, or Loosely Coupled Libraries

These are the lowest level, least integrated libraries; they are simply collections of named modules. These libraries, to be at all useful, must have the following attributes:

(1) Their constituent modules must be accessible by name by the appropriate translators. For example, a compiler must be able to read a source module directly from a source library and place its output directly in an object module library, without the intervention of data manipulation utilities.

(2) Utility programs that can convert modules in a library to other media must be available, as well as utilities to reclaim unused library space.

(3) The library management software must prevent the construction of modules with duplicate names, and must have safeguards against inadvertent module destruction. (For example, replacement of an "old" module by a "new" version with the same name should require an explicit replacement request, rather than be the default when a duplicate-name module is added to a library.)

(4) Basic programs to display the contents (module names) and unused space of a library must exist.

A.1.1.4.5.2 Integrated Libraries

Integrated libraries form a system of basic libraries whose module members are related and among which consistency is enforced. For example, the enforcement software should not allow a source module to be updated, edited or replaced unless an entry in the ECO (Engineering Change Order) Library exists that names the affected module. Furthermore, original modules are never replaced unless they are first archived to a long term storage medium. Minimum requirements of an integrated library system are as follows:

(1) Internal consistency between source modules, object modules, documentation, and test case streams must be enforced automatically.

(2) Hard copy audit listings must be produced for all changes.

(3) Archiving of old modules should be automatic and fail safe.

A.1.1.4.5.3 Automated Software Production and Test Systems

These tools add the following capabilities to integrated libraries:

(1) Inter-library and inter-module cross reference tables are automatically maintained. For example, in an integrated library system, a programmer still must manually determine all the affected modules to specify in an ECO that requires a change to a system subroutine. In an automated production and test system, the automatically produced and maintained cross reference tables will immediately show which modules reference the module to be changed.

(2) Furthermore, the inter-library links to the regression test library will indicate which tests must be run to verify that the change has had no unwanted side effects.

(3) The test case result library will indicate which result files will have to be altered or extended.

(4) The consistency checking software will then guarantee that a new system load module is not released until all required modifications to source code, documentation and test cases have been made; all compilations and linker runs executed; and all required regression tests successfully passed.

A.1.1.4.6 Performance Monitors

These tools assist the programmer in quantifying the resource consumption characteristics of a program, and in isolating performance critical areas. Performance monitors may be classified as either dependent on or independent from source language.

A.1.1.4.6.1 Language-Dependent Monitors

These tools allow the programmer to obtain "fine-grained" statistics on program execution (below the level of the program or subroutine) by inserting statements that instrument the program under test in order to obtain information such as the number times critical loops are executed, average depth of queue and list searches, numbers of calls to certain subroutines, and so on. These monitors are language dependent in the sense that they allow the programmer to insert "probes" into the source language program that must be compiled into calls to the monitor subroutines. Language-dependent monitors must provide the following functions:

(1) The ability to count event occurrences defined by the programmer, such as entries to subroutines, loop iterations, etc.

(2) The ability to obtain subroutine or loop timings. The timings must allow for the perturbations introduced by the monitor probe itself.

(3) The ability to maintain averaging counters by event and integrating counters over time.

(4) The ability to summarize a module's execution characteristics by percent of time spent in each program segment (WHILE or FOR loop, IF-THEN-ELSE leg, straight segment, closed subroutine) for a given set of test data.

A.1.1.4.6.2 Language-Independent Monitors

These tools allow the programmer to measure a module's performance characteristics in terms of its interactions with the other components of the system (operating systems, library routines, other user programs) with which it executes. This information must usually be gathered by activating probes that monitor the program's invocation of such operating system services as I/O requests, overlay requests, timer requests, and external module calls. Language independent monitors may also make use of CPU utilization figures maintained by the operating system for accounting or resource management purposes and on virtual and real memory utilization matters. They may also take advantage of the existence of special event recording hardware in the host processor, usually in conjunction with some degree of operating system support. Language-independent monitors should have the following capabilities:

(1) The ability to record the details of a module's utilization of system services such as I/O requests, timer service requests, and external calls that must be mediated by the supervisor.

(2) The ability to record typical execution versus wait time patterns that a program exhibits.

(3) The ability to record real storage utilization patterns.

(4) For demand paged virtual memory systems, the ability to monitor and record a module's instruction and data reference patterns over time, along with the ability to record its influence on paging rates.

A.1.1.4.7 Standards Enforcers

These tools allow source programs to be automatically examined, and checked for conformance to installation-defined standards of format, content, and usage. Since they must be able to recognize source language constructs, they are language-dependent. Standards enforcers should be able to perform at least the following functions, and the requirements that are to be enforced should be easily parameterized by the using installation:

(1) Check for binary conditions such as the presence of unwanted language constructs, or the absence of explicit data type declarations even when these conditions are permitted by the language.

(2) Check for quantitative conditions such as number of programs statements per module, ratio of comments to computational statements, or number of parameters per subroutine call.

(3) Allow the installation to define qualitative measures such as expression complexity.

A.1.1.4.8 Preprocessors/Reformatters

These tools assist programmers in producing well-structured and readable programs by allowing the introduction of well-structured programming constructs into source program for languages that do not have them (e.g., structured programming preprocessors for FORTRAN), and by automatically controlling the indentation of nested IF-THEN-ELSE clauses and loops, the placement of comments, and so on to produce readable listings. The use of these tools not only assists the production of more maintainable programs, but also increases programmer productivity by freeing the programmer from the need to observe strict columnization rules when preparing source module for input.

A.1.1.4.8.1 Preprocessors

Preprocessors should allow the definition of a simple structured programming constructs that produce suitable statements in the target source language. These should include the IF-THEN-ELSE, DO-WHILE, and CASE constructs at a minimum.

A.1.1.4.8.2 Reformatters

Reformatters should allow the using installation to flexibly define parameters for loop and IF-THEN-ELSE indentation, comment placement, and page ejection control. The reformatter should allow the source language programmer to make maximum use of printer lines that are longer than the image of the input statements.

A.1.2 Layer 2 Tools

A.1.2.1 Data Base Management Systems

These systems allow the user of a computer system to define the contents and the logical relationships between collections of data items that represent some useful abstraction of a real-world phenomenon (command and control functions, the modules and documentation of a system of computer programs) without being concerned with the physical mechanics of storing, locating and retrieving items or groups of items. While data base management systems have usually been designed to meet the needs of the commercial data processing market, they can provide a suitable base for a large, integrated software production facility and provide the framework for a large military system. In order to be a useful candidate for this role, a data base management system would have to possess the following characteristics:

(1) It should have the capability to store logically equivalent elements of varying size. It does not appear useful to attempt to manage program modules at the source statement level; therefore, the system must be able to store and maintain the logical relationships between varying sized units of source programs, documents, test case files, test result files, and so on.

(2) It must support dynamically varying structures, as different user requirements create the need to define different inter-unit relationships, different inter-module dependencies, etc.

(3) It should provide a suitable interface to the programs that must extract, modify and add data to the system, including language translators, linkers, editors, documentation aids, report writers, and project management aids.

(4) It should handle the problems of transaction journaling archiving, back up and recovery in such a way that the physical integrity of the data base is guaranteed.

(5) It must permit the logical consistency of the data to be easily maintained by user written utilities.

(6) It must have an effective security system that will prevent inadvertent or unauthorized access or destruction of data.

A.1.2.2 Project Management Aids

A.1.2.2.1 PERT/CPM Systems

PERT/CPM Systems assist managers in planning and controlling project activities. As a minimum, the system should provide two types of project elements (e.g., phases and tasks) and provide the following capabilities:

(1) Specification of personnel and equipment resource requirements at the task level.

(2) Specification of actual resource consumption at the task level.

(3) Project scheduling reports identifying potential slippages, earliest and latest initiation/completion times of tasks, and potential trouble areas.

(4) Project staffing reports identifying required and possible staffing levels.

(5)′ Equipment usage reports identifying required and possible needs correlated with staffing projections.

(6) Project status reports identifying accomplishments in relation to plan.

A.1.2.2.2 Project Estimation Systems

These systems assist in the develoment of work breakdown structures and related performance standards for use in estimating project resource requirements. The tools should have the following capabilities:

(1) Support multi-level work breakdown structures.

(2) Collection of performance data for multiple instances of a work breakdown structure.

(3) Develop and update performance standards from collected data.

(4) Compute statistical characteristics of the collected data to assist in evaluating the reliability of the standards.

(5) Provide reports to assist in identifying reasons for significant deviation of actual performance from the standard.

A.1.2.3 Documentation Aids

These tools assist the user in the preparation and maintenance of documentation about the modules of a system. Aids most relevant to a program development environment are as follows.

A.1.2.3.1 Text Processing Systems

These systems allow the maintenance of printed documents in machine readable form. In addition to the actual printed text of the document, the machine readable file generally contains control statements that specify the format of the finished document in terms of spacing, indentation, pagination, titling, and justification. More advanced systems may allow documents to be automatically constructed from fragments that can be dynamically combined at execution time, and can automatically construct indices and tables of contents. These systems must be combined with an online text editor to be useful.

A.1.2.3.2 Flowchart Construction Languages

These tools allow the construction of programs that can be compiled into printed or plotted flowcharts. The statements used to specify the flowcharts may usually either be kept in separate files or be embedded as comments in the source files of the programs whose logic flow they describe. This type of automated flowcharting has the advantage of allowing the programmer to specify the processing and decision blocks of the chart at an appropriate level of detail. It has the disadvantage of creating yet another "program" that must be maintained and that is as likely as not to become inconsistent with its associated source program.

A.1.2.3.3 Automatic Flowcharters

Automatic flowcharters use the source language programs themselves as input to construct a flowchart. This method has the obvious advantage of always being able to obtain a flowchart that is an accurate representation of the source module. It has the less obvious disadvantage of usually producing flowcharts that are specified at a greater level of detail than is desirable. There exists, of course, an overriding question of the utility of flowcharts as system documentation at any but the highest system block diagram level; many people feel that a well-documented structured program provides a more suitable reference.

A.1.2.4 Data Manipulation Utilities

These tools allow the system user to alter the format and content of data files independent of the logical significance of the data fields involved. While this category includes the standard media conversion utilities such as card-to-tape, tape-to-printer, etc., these have not been included because it is felt that simple programs of this type are almost universally available. The evaluation of tools in this category will therefore be restricted to sort/merge programs and editors.

A.1.2.4.1 Sort/Merge

This utility allows the user to rearrange the order of the logical records in one or more input files so that the records of the resulting output file are in order specified by the collating sequence of a series of one or more fields in each record. A usable sort/merge program should possess the following capabilities:

(1) It should process multiple input files.

(2) It should permit sorting and merging on multiple fields.

(3) It should allow a mixture of ascending and descending sequences to be specified for different fields in the same run.

(4) It should allow specification of any meaningful data type for a sorted field (e.g., binary, ASCII, floating point, etc.).

(5) It should allow use of any available checkpoint/restart facilities.

(6) It should permit exits to be taken to user programs at appropriate points (e.g., for end of volume processing, or to edit and delete individual input or output records).

A.1.2.4.2 Editors

Editors are programs that allow the user to add, delete, replace and alter the contents of individual data records within a file. They may be divided into interactive editors that are designed to be used in an on line, time-sharing mode, and batch editors that take their commands from control cards in the input stream. Both types of editors may be further subdivided into those that are primarily intended for use in developing source programs, and those that are designed to alter or "patch" binary object or load modules. The required features of each type of editor are summarized below.

A.1.2.4.2.1 Interactive Source Language Editors

These editors allow the user at a CRT or typewriter terminal to examine and alter source language statements. There exists a wide variety of editors in this category, and the features cited below are intended to represent the minimum functional requirements of a useful source editor:

(1) Must be able to locate items both by content (string comparison) and line number.

(2) Must be able to restrict the field of string search (and alteration) to be within fixed columns (to avoid altering sequence or statement numbers).

(3) Must be able to make global (file wide) changes of one string to another.

(4) Default actions and abbreviations should be fail-safe. For example, one well-known editor interprets a carriage return after a null line as a request to delete the data record last displayed. This is a very poor practice.

(5) Must be possible to save and retrieve intermediate edit sessions in case of system failure.

A.1.2.4.2.2 Interactive Object Module Editor

This editor can be used to alter binary object or load modules by inserting patches. Its purpose can be served by the source editor if that program has an option that allows a file's contents to be displayed and altered in hexadecimal or octal from a terminal.

A.1.2.4.1.3 Batch Source Language Editors

These editors usually work on a line number basis, using information supplied in control cards to add, delete or replace records in a particular file. The usability and human factors suitability of the programs vary widely, but, as a minimum, a batch editor must:

(1) Allow insertion, replacement, deletion and renumbering of the source file.

(2) Print a log of its actions that shows all alterations to the file, and provides enough line number information to allow future changes without obtaining a complete new listing.

A.1.2.4.1.4 Batch Object Module Editors

These binary patching programs should work either on a card sequence number (if it exists in the object library) or on a relative byte or word address within the module. Furthermore, the editor should have the ability to verify that the current contents of the locations to be changed match a hexadecimal or octal string supplied by the user prior to altering the locations to its new value. A change log should also be produced.

A.1.2.5 Information Retrieval Systems

These systems are general purpose application programs operating either on-line (interactively) or in the batch that interpret user requests to locate and display information that is stored either within a structured data base or within separate files. These systems can be classified either as query language systems or as report writers.